Universidad de Alicante
Departamento de Lenguajes y Sistemas Informáticos

Curso de Algoritmia Avanzada

Rafael C. Carrasco

Actualizado el 26 de septiembre de 2006 Version 2.0

Presentación

Este documento pretende servir de material básico para un curso de *Algoritmia Avanzada*, por lo que presupone conocimientos elementales de programación, diseño y análisis de algoritmos, estructuras de datos y teoría de la probabilidad.

Para conseguir un aprendizaje más efectivo, se ha incluido textos interactivos durante cuya lectura el lector puede (y debe) tomar decisiones. Este diseño está inspirado en la filosofía *constructivista* desarrollada por Piaget y otros autores (véase, por ejemplo, J. Novak: A Theory of education. Cornell University Press, 1977.)

Advertencias

Para seguir este curso es preciso utilizar un ordenador personal con Acrobat Reader™ 5.0 (o superior) y un compilador de `Java` o `C++`.
El documento contiene ejercicios y problemas: consulta la solución de cada ejercicio sólo después de haber escrito tu respuesta; los problemas deben ser revisados por el profesor.
Los enlaces se distinguen en color verde. Aprieta aquí si quieres ajustar el tamaño del texto.

Se ha optado por usar el punto para separar la mantisa y la parte entera de los números reales para facilitar la lectura de las listas de números (por ejemplo, los argumentos de una función) que se separan mediante comas.

Índice

1. Programación dinámica

La programación dinámica debe su nombre a los problemas de dinámica de sistemas a los que se aplicó originalmente. Su característica más destacable es que aumenta considerablemente la eficiencia de numerosos procedimientos recursivos.

1.1. Programación dinámica recursiva

Ejercicio 1.

(a) El *Nim* es un juego de estrategia en el que cada jugador debe retirar alternativamente entre una y N fichas de la mesa. Pierde el jugador que no tiene opciones válidas en su turno, bien porque no quedan fichas o bien porque encuentra una pero el jugador anterior retiró también una. Diseña una función recursiva $f(m, n)$ para el Nim de dos jugadores y $N = 3$ que sea positiva si existe una estrategia ganadora para el jugador que encuentra m fichas sobre la mesa si el jugador anterior retiró n fichas (y negativa en caso contario).

(b) Compara el tiempo de cálculo de la función anterior para $m = 40$ y para $m = 50$. ¿Para qué valores de n es útil el algoritmo?

(c) ¿Cuál es la causa del crecimiento exponencial en función de m del coste del cálculo? Ayuda: dibuja el árbol de llamadas recursivas que se genera para $f(6, 0)$.

(d) Modifica el algoritmo para evitar, en la medida de lo posible, el aumento de llamadas a la función f.

La esencia de la programación dinámica es utilizar un almacén para que el programa recursivo guarde resultados intermedios que pueden ser reutilizados para acelerar el cómputo. A esta técnica la llamaremos programación dinámica recursiva (en inglés, "memoization").

Ejercicio 2.

(a) ¿Cuál es el valor límite de m para el que se puede calcular f con la nueva versión del algoritmo?

(b) Justifica que el programa (4) no ejecuta más de $\mathcal{O}(M)$ pasos.

Problema 1. Implementa en `Java` un programa que juegue al Nim. Para ello, modifica la función para que el valor de retorno sea el número de fichas que debe retirarse para ganar o un valor negativo si no existe estrategia ganadora.

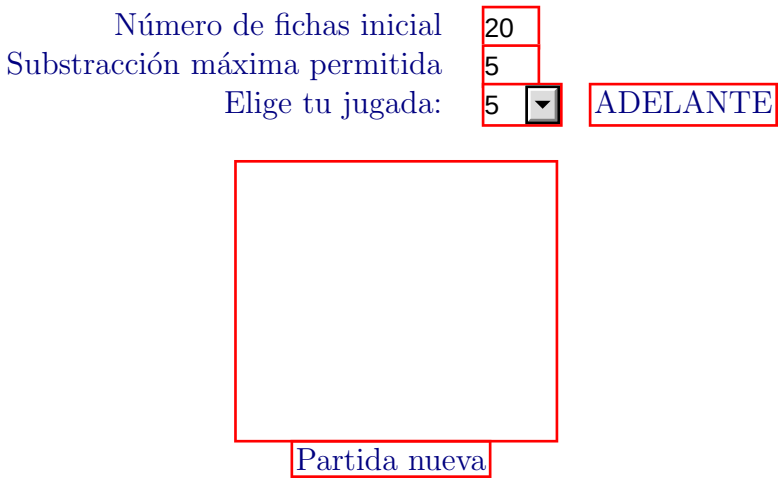
1.2. Programación dinámica iterativa

La programación dinámica iterativa suele resultar más rápida y además evita el coste de memoria asociado a las llamadas recursivas. Por contra, a veces calcula más posiciones del almacén de las necesarias, por lo que no es siempre más rápida que la programación dinámica recursiva.

Ejercicio 3. Implementa una versión iterativa que evite el desbordamiento de la pila recursiva en el algoritmo (4) y compara la velocidad y el número de elementos del almacén calculados.

Ejercicio 4. ¿Cómo podemos evitar que un algoritmo de programación dinámica iterativa calcule más elementos de los que necesita la versión recursiva?

Problema 2. Generaliza el porgrama para que èrmita jugar a varios jugadores. Diseña un formulario en `HTML` similar al siguiente que utilice el programa en `Java` implementado anteriormente (con las modificaciones que sean necesarias).



1.3. La mayor subsecuencia común

La orden `diff` de Unix compara las líneas de dos ficheros A y B e informa sobre las diferencias encontradas entre ambos. Para ello aplica una función de dispersión ("hashing") que transforma cada línea en un único entero. Posteriormente busca la mayor subsecuencia común a las secuencias obtenidas, esto es, qué líneas deben borrarse de cada uno para obtener dos ficheros idénticos.

Ejercicio 5.

(a) ¿Qué operaciones deben realizarse para transformar la secuencia "nuclear" en "unclear"? ¿Cuál es la longitud de la mayor subsecuencia común?

(b) ¿Cuál el la m.s.c. de las cadenas "algoritmia" y "avanzada"? Une mediante líneas cada par de letras iguales que forma parte de la m.s.c.

(c) Analiza en qué casos es trivial comparar dos ficheros por líneas. Escribe una fórmula recursiva para los casos más complejos pensando en qué posibilidades hay 1) cuando las dos últimas líneas son distintas y 2) cuando son iguales.

(d) Calcula la mayor subsecuencia común de las dos cadenas siguientes: 01001100011100001111 y 11110011011000100011.

(e) ¿Cuántas casillas de la tabla se calculan cuando se aplica programación dinámica recursiva al ejemplo anterior? ¿Y en el caso iterativo?

(f) ¿Cuál es el coste temporal (en el peor caso) de computar la mayor subsecuencia común de dos cadenas A y B usando programación dinámica? ¿Y el de obtener la de tres cadenas A, B y C?

El problema de la mochila discreto se plantea de la siguiente forma: dados N objetos de pesos $w_1, \ldots, w_N$ y valores $v_1, \ldots, v_N$, calcúlese el valor máximo de los objetos que se pueden transportar en una mochila de capacidad M.

Ejercicio 6.

(a) Escribe una fórmula recursiva para el valor máximo $f(n, m)$ que se puede transportar en la mochila sin sobrepasar el peso m tomando sólo objetos entre los n primeros.

(b) El problema de la mochila discreto es un ejemplo de problema de complejidad asintótica exponencial al que se puede aplicar programación dinámica. Si los pesos son enteros, el coste temporal está en $\mathcal{O}(NM)$ siendo N el número de objetos y M la capacidad de la mochila. ¿Significa lo anterior que hemos encontrado un algoritmo polinómico para un problema exponencial?

(c) Calcula una solución óptima y dibuja las posiciones de la tabla calculadas en el caso con $N = 10$, $M = 40$ en el que pesos y valores coinciden y son los siguientes: 23, 6, 17, 35, 33, 15, 26, 12, 9 y 21.

(d) Explica cómo evitar el coste proporcional a NM cuando $2^N \ll M$.

Problema 3. Diseña un programa que:

- Extraiga de una colección de textos las frecuencias de las palabras que en ellos aparecen.
- Calcule para una secuencia de números (procedente del teclado de un teléfono móvil) cuál es la palabra más probable asociada a dicha secuencia.

2. Ramificación y poda

Los algoritmos de ramificación y poda pueden entenderse como una extensión de los algoritmos de retroceso. Mediante esta técnica se pueden tratar problemas de optimización difíciles (incluso de la clase NPH) y encontrar soluciones, al menos aproximadas, para estos problemas.

2.1. Búsqueda mediante ramificación del dominio

Supongamos que queremos obtener el valor máximo de la función $\phi(x) = xe^{-x}$ en el intervalo de números reales $D = [0, 6]$. Por supuesto, hemos olvidado nuestros conocimientos anteriores sobre análisis de funciones.

Llamaremos solución a todos los elementos del dominio D. A la función ϕ que evalúa la calidad de las soluciones contenidas en D la llamaremos función objetivo.

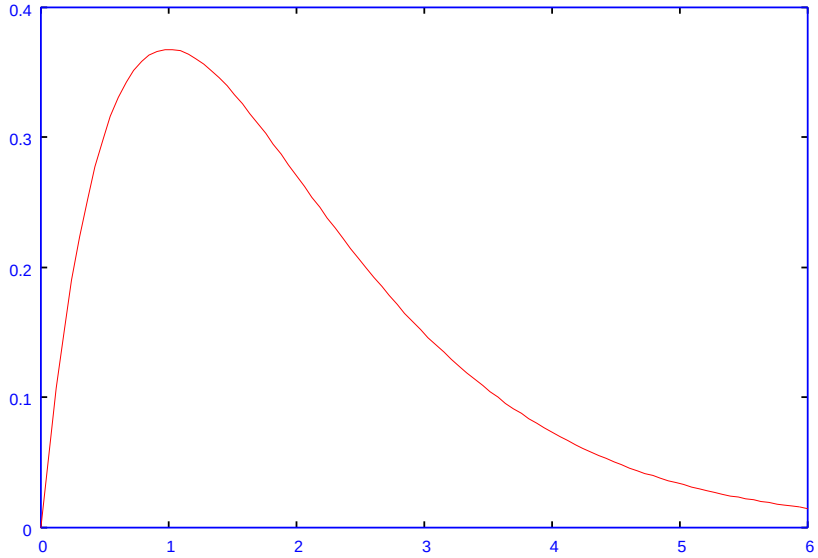


Figura 1: La función $x \exp(-x)$.

Un algoritmo sencillo para calcular el máximo de ϕ es el siguiente:

```
double epsilon = 1e-4; // precisión de búsqueda

double f ( double xmin, double xmax ) {
 double xmid = ( xmin + xmax ) / 2;
 if ( xmax - xmin < epsilon )
  return std::max ( xmin * exp(-xmin), xmax * exp(-xmax) );
 else
  return std::max ( f (xmin, xmid), f (xmid, xmax) );
}
```

Algoritmo 1

En este ejemplo, la función $f(X)$ calcula el valor óptimo de la función objetivo ϕ en cualquier intervalo $X = [a, b]$. En general, definiremos $f(X)$ como el valor óptimo de la función objetivo ϕ en el subconjunto X, esto es, $f(X) = \max_X \phi(x)$.

El algoritmo anterior sigue una estrategia del tipo divide y vencerás aplicada al dominio de la función objetivo: el dominio de búsqueda se subdivide en dominios más pequeños para encontrar el valor óptimo en cada subdominio y, mediante comparación de los resultados, el valor óptimo global. Esta forma de proceder es típica de los algoritmos de ramificación y poda (llamados "branch and bound" en inglés).

Si el valor óptimo de ϕ en D es $\texttt{ybest} = f(D)$, un elemento `xbest` del dominio D es una solución óptima si $\texttt{ybest} = \phi(\texttt{xbest})$.

Observa el algoritmo 1 puede servir también para funciones con múltiples máximos locales, simplemente cambiando la definición de la función.

El valor óptimo de la función objetivo ϕ puede alcanzarse para más de una solución. En ese caso, no podemos hablar de solución óptima sino de soluciones óptimas.

Ejercicio 7.

(a) Implementa el algoritmo de búsqueda en forma de un programa en lenguaje `Java`. ¿Cuántas llamadas recursivas realiza para calcular $f(0, 6)$?

(b) Representa mediante un árbol las llamadas recursivas que realiza el algoritmo tomando $\epsilon = 1.0$ en vez de $\epsilon = 10^{-4}$.

> Al árbol generado por un algoritmo de ramificación y poda se le suele denominar árbol de estados.

(c) Justifica si es cierta o no la siguiente afirmación: en un problema de maximización como el anterior, si $X = \bigcup_k Y_k$, entonces $f(X) = \max_k f(Y_k)$.

2.2. Funciones de cota optimista

Una cota optimista de f es cualquier función sencilla que aplicada al subconjunto X nos proporciona un valor $g(X)$ mejor o igual que el que obtendremos al aplicar f al mismo subconjunto. Por tanto, en un problema de maximización, toda cota optimista satisface $g(X) \geq f(X)$.

Test. ¿Cuáles de las siguientes funciones son cotas optimistas del valor máximo de $\phi(x) = xe^{-x}$ en el intervalo $[a, b]$.

(a) ae^{-a}. (b) be^{-b}. (c) be^{-a}. (d) ae^{-b}. (e) b.

El siguiente ejercicio muestra la utilidad de conocer funciones de cota optimista.

EJERCICIO 8. Supongamos que se ha calculado $f(0,3)$ y que nuestro algoritmo procede a calcular $f(3,6)$:

(a) ¿Cuánto vale $g(3,6)$ si $g(a,b) = be^{-a}$?
(b) Teniendo en cuenta el valor anterior y que $f(0,3) = 0.37$, ¿qué podemos deducir con respecto al resultado $f(0,6)$?
(c) ¿Es posible llegar a la misma conclusión si $g(a,b) = b$? ¿Son todas las cotas optimistas igual de útiles?

Recuerda: si en un problema de maximización se cumple $g_1(X) < g_2(X)$, entonces g_1 es mejor cota optimista que g_2 en el subdominio X. En general, las cotas superiores son mejores cuanto más bajas (por supuesto, siempre y cuando sean cotas).

Ejercicio 9. Reescribe el algoritmo 1 para que evite el cálculo de f cuando la cota optimista sea peor que el mejor resultado obtenido hasta el momento. Para ello utiliza una variable global llamada `ybest` que almacene el mejor valor de ϕ obtenido hasta ese momento, lo que permite definir f de tipo `void`.

La acción y el efecto de evitar la exploración de zonas del dominio mediante la comparación de funciones de cota optimista con el mejor valor encontrado hasta el momento recibe el nombre de poda.

Ejercicio 10.

(a) Dibuja las llamadas a f realizadas por la nueva versión cuando se calcula $f(0, 6)$ con ϵ =1.0.

(b) Supongamos que en el algoritmo anterior intercambiamos el orden de las llamadas a `f(xmin,xmid)` y `f(xmid,xmax)`. ¿Cómo cambia la eficiencia de la poda?

(c) En general, ¿qué tipo de mejoras pueden hacerse para aumentar la eficiencia de la poda?

2.3. Funciones de cota pesimista

Como hemos visto, la eficiencia de la poda viene determinada en gran medida por la calidad de las soluciones provisionales encontradas (y almacenadas) durante la ejecución del algoritmo. Mejorar la eficiencia es esencial cuando el dominio de búsqueda es muy extenso (de hecho, podemos apostar que así será el dominio en la mayoría de los problemas que se resuelven mediante ramificación y poda).

Una forma de mejorar la eficiencia de la poda es calcular soluciones aproximadamente óptimas en cada nodo visitado del árbol de estados.

> Llamaremos cota pesimista $h(X)$ a cualquier función sencilla que calcula el valor de la función objetivo ϕ para una solución $\hat{x}$, no necesariamente óptima, de X.

En un problema de maximización, el valor de $f(X)$ no puede ser menor que el de $h(X)$ pues bien $h(X)$ es el valor óptimo y entonces $h(X) = f(X)$ o bien $h(X)$ es subóptimo y $h(X) < f(X)$.

En un problema de maximización, el valor de $h(X)$ acota inferiormente el valor de $f(X)$ mientras que $g(X)$ lo hace superiormente, esto es, $h(X) \leq f(X) \leq g(X)$

Test. De acuerdo con lo anterior, ¿cuáles de las siguientes son funciones de cota pesimista $h(a, b)$ para $\phi(x) = xe^{-x}$?

(a) ae^{-a}.

(b) be^{-a}.

(c) be^{-b}.

(d) ae^{-b}.

(e) máx(ae^{-a}, be^{-b}).

Ejercicio 11. Dibuja sobre cada nodo del árbol de llamadas recursivas del ejercicio 10 los valores de f, g y h.

Ejercicio 12. Reescribe el programa de búsqueda para que cada llamada a la función $f(a, b)$ actualice el valor de la variable `ybest` con el resultado $h(a, b)$.

El tiempo de cálculo de f en un nodo hoja del árbol de estados (esto es, sin descendencia) suele ser pequeño. Por ello, conviene que el valor de las cotas y el de la función objetivo coincidan en los nodos hoja.

Ejercicio 13. Incluye en tu programa un contador del número de llamadas a la función f que se generan durante el cómputo de $f(0,6)$ y completa la siguiente tabla con el número de llamadas en la versión original (1), cuando se incluye la poda (6) mediante g y cuando se incluye además la actualización (7) de `ybest` mediante h.

	$\epsilon = 1$	$\epsilon = 10^{-4}$	$\epsilon = 10^{-6}$
simple			
con g			
con h			

En los algoritmos de *ramificación y poda*, el uso combinado de funciones de cota optimista y pesimista permite reducir considerablemente el tiempo de búsqueda.

Un error frecuente al diseñar cotas pesimistas consiste en buscar funciones $h(X)$ peores que todos los valores de ϕ en X. Esto conduce a cotas demasiado pesimistas, bastante alejadas de f y, por tanto, poco útiles para la actualización de `ybest`.

Para comprender mejor estas propiedades, consulta el apéndice A.

2.4. Cálculo explícito de la solución

Ejercicio 14. Reescribe el programa para que calcule una solución óptima `xbest` además del valor óptimo `ybest`=ϕ(`xbest`). Modifica el programa y las funciones de cota para calcular la solución óptima de $\phi(x) = x\exp(-\frac{x}{10}) + \sin(x)$ en el intervalo $[0, 20]$.

2.5. Ramificación mediante estrategias inteligentes

En un procedimiento recursivo, cada llamada a la función permanece activa mientras se resuelven los cómputos recursivos generados a partir de ella. Por ello, llamaremos nodos activos a aquellos nodos cuyo cómputo ha sido iniciado pero no ha sido completado.

Los nodos activos más aquellos que aún no han sido explorados se denominan nodos vivos. Por contra, los nodos podados o cuyo cómputo ha finalizado se denominan son nodos muertos.

Ejercicio 15. Señala en el árbol del ejercicio 11 qué nodos son activos, vivos y muertos cuando se está calculando $f(0.75,1.5)$.

Ejercicio 16.

(a) Reescribe el algoritmo de ramificación y poda 7 para que en vez de proceder de forma recursiva ciega, elija en cada momento el intervalo vivo $[a, b]$ para el que la cota optimista $g(a, b)$ sea mayor. Para ello, utiliza una simulación iterativa de la recursión con un sólo nodo activo en cada instante.

(b) Compara el número de iteraciones de este algoritmo con el número de llamadas de la mejor versión recursiva. ¿Cuándo conviene utilizar estrategias inteligentes en vez de ciegas?

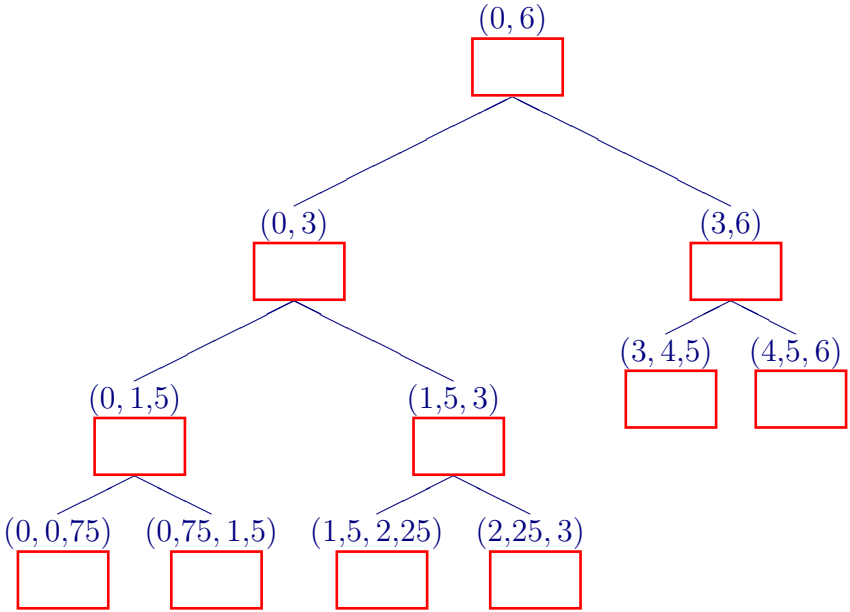
- **Demostración de ramificación y poda**

La siguiente animación permite entender cómo cambia en cada iteración del algoritmo 10 la lista de nodos vivos (con fondo blanco), nodos activos (con fondo amarillo) y nodos muertos (con fondo gris). Para ello se ejecutan reiteradamente los siguientes pasos:

1. Activación del siguiente nodo X y cómputo de su cota pesimista $h(X)$. Actualización de `ybest`, si `ybest` $< h(X)$.

2. Expansión de los hijos del nodo activo y cómputo de sus cotas optimistas.

Inicio Avance ybest=]



Observa en el ejemplo anterior que algunos nodos como el $(3, 6)$ se activan a pesar de que su cota optimista ha quedado por debajo de `ybest`, por lo que no cabe esperar soluciones mejores en esas ramas.

EJERCICIO 17. En los programas iterativos de ramificación y poda es posible eliminar nodos vivos tras cada actualizacion de `ybest`. Añade una poda adicional en el algoritmo que elimine de la lista todo nodo vivo no explorado X que deje de cumplir la condición $g(X) >$ `ybest`.

Ejercicio 18. El recorrido recursivo del árbol de estados visita los nodos en preorden (es preciso pasar por el padre para llegar a los hijos). Sabemos que se pueden utilizar otras estrategias ciegas como el recorrido en anchura. ¿Cuáles son las ventajas e inconvenientes de cada una de estas opciones? ¿Es posible realizar podas en un recorrido en postorden?

En la búsqueda mediante ramificación y poda se puede optar por una estrategia de recorrido ciega (esto es, independiente del problema), inteligente (es decir, basada en los datos) o mixta (combinación de ambas).

2.6. Minimización de funciones

Ejercicio 19. Revisa los algoritmos diseñados hasta ahora y señala qué cambios habría que hacer para obtener una solución óptima que minimiza una función dada $\varphi(x)$.

2.7. Ramificación y poda con restricciones

En algunos problemas, no todos los elementos del dominio satisfacen las especificaciones. En ese caso, llamamos solución sólo a aquellos elementos x que cumplen la condición $c(x)$. La solución óptima es, por tanto, la mejor de las soluciones según la función objetivo ϕ.

EJERCICIO 20. ¿Cuál debe ser el valor de $f(X)$ si $X \subset D$ es un subdominio que no contiene soluciones?

Si $X = \emptyset$ o X no contiene soluciones, entonces tanto $f(X)$ como $h(X)$ deben tomar un valor pésimo. Aunque la función $g(X)$ puede tomar cualquier valor conviene que, en este caso, $g(X) = f(X) = h(X)$.

A continuación sigue un ejemplo.

Ejercicio 21. Una empresa de M trabajadores desea presentar N productos en una convención enviando por cada producto n al menos a uno de los trabajadores involucrados en su desarrollo. Cada trabajador m ha participado en un número de productos C_m y la empresa quiere minimizar el número total de trabajadores enviados. Si las soluciones se representan mediante vectores binarios de M componentes $(x_1, \dots x_M)$ y T_{mn} es 1 si m ha participado en n y 0 en caso contrario, escribe una función de cota optimista y otra pesimista para el estado $X = (x_1, \dots, x_m, *, \dots, *)$.

Ejercicio 22. Responde a las siguientes cuestiones:

Inicio del Test

1. La diferencia fundamental de un algoritmo de ramificación poda con uno retroceso es:

- ☐ El uso de cotas para podar.
- ☐ El uso de cotas pesimistas para actualizar el mejor valor.
- ☐ El uso de recorridos inteligentes.
- ☐ El retroceso sólo sirve para problemas de optimización.

2. En un problema de *minimización* conviene que la cota optimista g sea

- ☐ Lo menor posible.
- ☐ Lo mayor posible siempre que sea menor que f.
- ☐ Mayor que f y lo más cercana a f posible.

3. En un problema de minimización, si Y es un subconjunto de X:

- ☐ Se cumple siempre $f(X)$ es menor o igual que $f(Y)$.
- ☐ Conviene que $g(X) \geq g(Y)$.
- ☐ Podemos exigir que $h(X)$ sea mayor o igual que $h(Y)$.

4. En un caso de minimización, la cota pesimista $h(X)$

- ☐ Siempre será menor que cualquier otro valor en X.
- ☐ Será menor que el de la mejor solución en X.
- ☐ Puede ser mayor o menor que el de otros valores de ϕ en X.

5. ¿Cuál de las siguientes afirmaciones es cierta?

- ☐ En un problema de minimización, es preferible elegir antes la rama con cota optimista menor.
- ☐ En cualquier caso, es preferible elgir la rama cuya cota optimista sea más ajustada (p.ej., más baja en maximización).
- ☐ En un árbol binario, es posible utilizar funciones de poda haciendo un recorrido en intraorden (subárbol izquierdo, padre, subárbol derecho).
- ☐ Si $g(X) = g(D)$, entonces X contiene una solución óptima del dominio D.

6. ¿Cuál es el tamaño máximo de la lista en el programa inteligente 10 con $\epsilon = 10^{-6}$?

- ☐ Menos de 1000.
- ☐ Entre 1000 y 2000.
- ☐ Entre 4000 y 5000.
- ☐ Entre 90.000 y 100.000.

Final del Test Puntos:

3. Búsqueda de texto

La importancia de utilizar métodos eficientes para localizar información textual es mayor desde la popularización de internet y de las bibliotecas digitales. A veces, es posible construir índices (por ejemplo, Google); otras veces, la búsqueda ha de realizarse sin preprocesamiento.

3.1. Búsqueda de texto sin preprocesamiento

Si la extensión de los documentos es moderada (por ejemplo, inferior a un Megabyte) es posible utilizar algoritmos de búsqueda simples. El más sencillo, aunque menos eficiente, es la fuerza bruta:

```
void find ( string P, string T ) {
  int i, j, m = P.length(), n = T.length();
  for ( j = 0; j + m <= n; ++j ) {
    for ( i = 0; i < m && T[j+i]==P[i]; ++i );
    if ( i == m )
      std::cout << "Match: " << j << endl;
  }
}
```

Algoritmo 2

Figura 2: Búsqueda del patrón P en el texto T mediante fuerza bruta.

El procedimiento de la fuerza bruta puede representarse gráficamente como una regla P que se desliza una posición cada vez sobre otra regla fija T hasta que P coincide con las casillas de T que cubre.

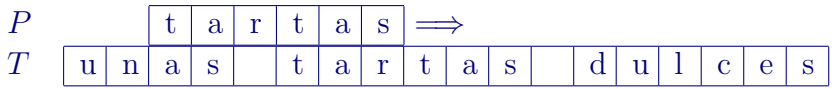


Figura 3: Representación gráfica de la búsqueda por fuerza bruta.

El coste de esta búsqueda es proporcional al producto mn de las longitudes.

El algoritmo de Aho y Corasick

El procedimiento de Aho y Corasick construye un autómata finito determinista para procesar el texto, tal y como se representa en la siguiente figura. Si

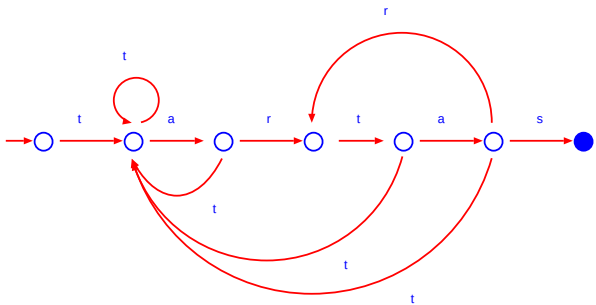


Figura 4: Autómata de Aho y Corasick para la cadena “tartas”.

el autómata llega al estado de aceptacioń (marcado en oscuro) significa que ha encontrado un caso del patrón. La búsqueda tiene en este caso un coste proporcional a n, pero el de construcción del autómata es proporcional a m y al tamaño del alfabeto.

El algoritmo de Boyer y Moore

El procedimiento de Boyer y Moore toma una decisión a primera vista sorprendente: aunque desliza el patrón sobre el texto de izquierda a derecha, compara el patrón con el texto de derecha a izquierda.

Ejercicio 23.

(a) Supongamos que el inicio del patrón P pasa de la posición $j-1$ del texto a la j para la siguiente comparación y que $T[j+i]$ es un carácter que no aparece en ningún sitio de P (toma como ejemplo $j=0$, $i=5$, $P=$ "tartas" y $T=$ "tartana"). ¿Cuánto podemos incrementar con seguridad j en la siguiente iteración?

(b) ¿Cuál es la ventaja de empezar a examinar la posición $T[j+m-1]$?

Ejercicio 24.

(a) Supongamos ahora que el carácter $T[j+i]$ aparece en P en la posición $k < m$ (por ejemplo, al comparar $P =$ "tartas" y $T =$ "carpetas" con $j = 0$) ¿Cuánto podemos incrementar ahora j?

(b) ¿Qué valor de k debemos asignar al caso en el que el carácter no está en P (como en el primer ejemplo) para que sea un caso particular del anterior?

Supongamos que estamos comparando $P =$ “tarta” con $T =$ “gratamente”. Cuando $j = 0$ y $i = 2$, encontramos que $T[j + i] =$ “a” no coincide con $P[i] =$ “r”. Sin embargo, el criterio anterior no permite desplazar P ya que $k = 4$ para la letra a y, por tanto, $i - k = -2$. Para solventar esta dificultad debemos desplazar P a la derecha máx$(1, i - k)$ posiciones.

> Cuando la última posición en P del carácter discrepante se encuentra a la derecha de la posición actual (esto es, $k > i$), la variable j debe aumentarse, al menos, en uno.

Ejercicio 25. Implementa una función que calcule el valor del desplazamiento para cada carácter y explica cómo debe modificarse el algoritmo de búsqueda por fuerza bruta para aprovechar esta información.

El algoritmo original de Boyer y Moore (1977) incluía dos estrategias de desplazamiento:

1. Mal carácter (la descrita anteriormente).
2. Buen sufijo.

La estrategia del buen sufijo es más compleja de programar (de hecho, su implementación fue publicada posteriormente) y su eficacia es menor.

La idea del buen sufijo trata de aprovechar la información obtenida de los caracteres coincidentes. En el ejemplo siguiente, el sufijo "ta" aparece como prefijo de la subcadena que se inicia en $P[0]$, lo que permite desplazar el patrón tres posiciones a la derecha.

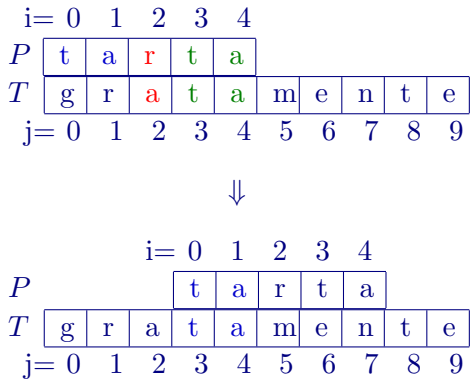


Figura 5: Desplazamiento por buen sufijo.

Ejercicio 26.

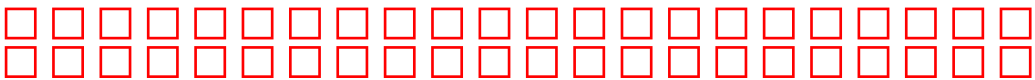
(a) ¿Cuál es el desplazamiento por buen sufijo adecuado al comparar $P =$ "domado" y $T =$ "vedado"? Describe las comprobaciones y desplazamientos que deben realizarse en un caso general.

(b) Implementa una función que calcule los desplazamientos válidos para cada sufijo de P y modifica el algoritmo de la página 173 para aprovechar esta información.

- **Demostración del algoritmo de Boyer y Moore**

Escribe cualquier patrón P y cualquier texto T en las casillas correspondientes. Cada vez que aprietes el botón NEXT aparecerán marcados en verde los caracteres coincidentes (y el siguiente sufijo idéntico en P con borde rayado) y en rojo el primer carácter discrepante (y en azul el próximo carácter que coincide en P).

Patrón: tomatito
Texto: maragatos



Desplazamientos propuestos:
por mal carácter
por buen sufijo

NEXT

3.2. Búsqueda de texto con preprocesamiento

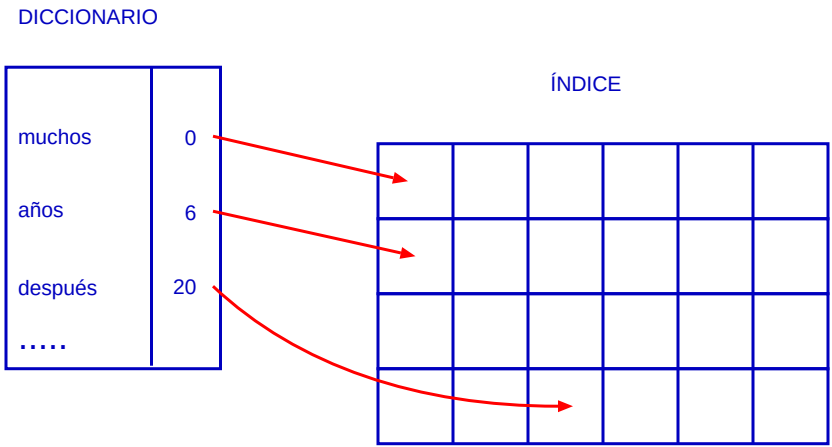
En las colecciones de tamaño considerable (por ejemplo, un Gigabyte) no es posible realizar búsquedas secuenciales. Por supuesto, mucho menos en el caso de internet. Para este fin se suelen utilizar técnicas de indizado: registrar para cada palabra sus lugares de aparición.

Una consecuencia de la ley de Zipf es que el tamaño del léxico usado crece casi linealmente con el tamaño de los textos. Por ejemplo, para colecciones de 10^5 a 10^8 palabras tamaño del léxico es aproximadamente una décima parte del texto.

Ejercicio 27. Para colecciones medianas (hasta 4GB), 32 "bits" pueden servir para identificar cada caso de una palabra. ¿Cuál será aproximadamente el tamaño del índice de una colección de textos de 1GB?

Los diccionarios que contienen el léxico utilizado pueden ser almacenados normalmente en la memoria de acceso directo. El índice propiamente dicho suele tener un tamaño comparable a la colección y residirá en la memoria de acceso indirecto.

Para minimizar la memoria utilizada guardaremos el índice como un vector en el que cada posición contendrá la dirección (absoluta dentro de la colección) del inicio de una palabra de la colección. El intervalo de posiciones asociado a una palabra estará almacenada en el diccionario.



Observa que basta con guardar en el diccionario el límite inferior de cada intervalo: el superior puede deducirse restando uno al inferior de la siguiente palabra.

Además del diccionario de palabras, necesitaremos otra tabla con los nombres de los documentos indizados y sus tamaños.

Fichero	Offset
Cap1.xml	127
Cap2.xml	234
Cap3.xml	333
⋮	⋮

Ejercicio 28. Implementa una clase en `Java` que construya dos tablas asociativas: una con el número de veces que aparece cada palabra en una colección y otra con el nombre absoluto de los ficheros y sus tamaños.

Ejercicio 29. Implementa una función para la clase anterior que, dadas las tablas, escriba el vector índice en un fichero binario.

Para bibliotecas de tamaños cercanos al Gigabyte, los diccionarios pueden ser alojados en la memoria de acceso directo, por lo que su construcción es rápida (del orden de 10 minutos incluyendo la descarga secuencial al disco). La creación del índice requiere descargar información (aleatoriamente) al disco duro pero, si se evita la sincronización, el coste es similar.

Ejercicio 30. Implementa una función tal que, dada una palabra, devuelva un vector con las posiciones absolutas de sus apariciones dentro de la colección (hasta un límite) y otra que dada una direccón absoluta devuelva el nombre del fichero y la posición relativa al primer carácter del fichero.

Problema 4. Diseña un formulario (y las funciones adecuadas) que muestre los contextos donde aparece una palabra. Procura que los diccionarios sean compartidos por todos los procesos de búsqueda.

El inconveniente de los índices es que no permiten resolver con eficiencia la búsqueda de frases porque es difícil determinar si dos palabras son adyacentes o no sin postprocesamiento (además de que se pueden generar muchos resultados intermedios inútiles).

Para la búsqueda de expresiones o frases puede convenir el uso de cadenas de sufijos. Si quieres saber más sobre las cadenas de sufijos, puedes leer aquí.

4. Compresión

A pesar de la reducción del precio de la memoria digital, la cantidad de información que necesitamos almacenar es cada vez mayor (¿cuántos Terabytes contiene internet?) y el tiempo de transmisión sigue siendo caro. Por tanto, los algoritmos de compresión nos ayudan a reducir el consumo de tiempo y dinero.

Ejercicio 31.

(a) ACME Software asegura que su programa reduceIt es capaz de reducir el tamaño de cualquier fichero que se desee comprimir preservando toda la información en él contenida. ¿Es veraz esta afirmación? Ayuda: piensa en cuántos ficheros hay de tamaño menor o igual que 1 kilobyte.

(b) Aplica diversos métodos de compresión (`gzip`, `bzip2`, `zip`, etc.) a un fichero con "bits" aleatorios. ¿Cuál es la tasa de compresión obtenida?

4.1. Compresión basada en diccionarios

Muchos métodos de compresión (como `gzip` o `compress`) utilizan la información obtenida de la parte ya procesada del fichero para no reescribir cadenas que se repiten. Esta clase de algoritmos (Ziv y Lempel, 1978) usa un diccionario que asigna códigos numericos a las cadenas de forma similar a la siguiente:

1. Asígnese en el diccionario un código a la cadena vacía ϵ y hágase $w = \epsilon$.

2. Si no hay más símbolos en la entrada, escríbase el código de w (salvo que $w = \epsilon$) y termínese.

3. Léase un nuevo símbolo c de la entrada; si wc está en el diccionario, hágase $w = wc$ y vuélvase a 2; en caso contrario pásese a 4.

4. Escríbase el código de w seguido de c; añádase wc al diccionario (esto es, asígnesele un código); hágase $w = \epsilon$ y vuélvase al paso 2.

Este proceso puede observarse en el formulario de la página siguiente.

- **Demostración del algoritmo de Ziv y Lempel**

Input

Next

Output

Diccionario

0001 = ()

- Escribe una cadena de entrada en el campo “input”.
- El botón “next” muestra el contenido del diccionario y la salida paso a paso.

A diferencia de nuestro ejemplo, en la práctica se suele utilizar sólo ceros y unos como símbolos de entrada y salida (esto es, se utiliza la representación binaria de los ficheros) y códigos de unos 12 dígitos.

Ejercicio 32. ¿Qué podemos hacer si todos los códigos disponibles ya están asignados?

4.2. Compresión de Huffman

Algunos procedimientos de compresión, como `bzip2`, utilizan el método de Huffman. Esencialmente, este método consiste en asignar códigos de longitud variable a los símbolos del texto: más largos a los caracteres (o combinaciones de ellos) menos probables. Esta idea se usaba ya en el código Morse.

Para obtener los códigos se construye un árbol binario de la siguiente forma:

1. Por cada carácter c con frecuencia $f(c)$ créese un árbol con un único nodo etiquetado con $f(c)$.

2. Mientras haya más de un árbol, tómense los dos árboles de menor frecuencia y agrúpense como hijos de un nuevo nodo; la frecuencia del árbol resultante es la suma de las frecuencias de los hijos.

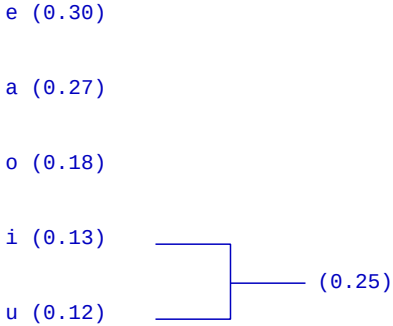
- **Demostración del algoritmo de Huffman**

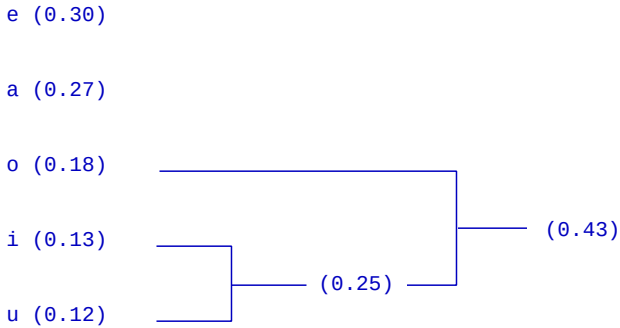
A continuación se ilustra la construcción del árbol para las vocales del castellano, cuyas frecuencias relativas son aproximadamente las siguientes:

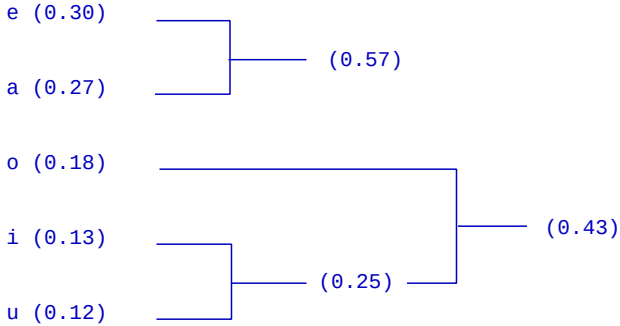
a	e	i	o	u
0.27	0.30	0.13	0.18	0.12

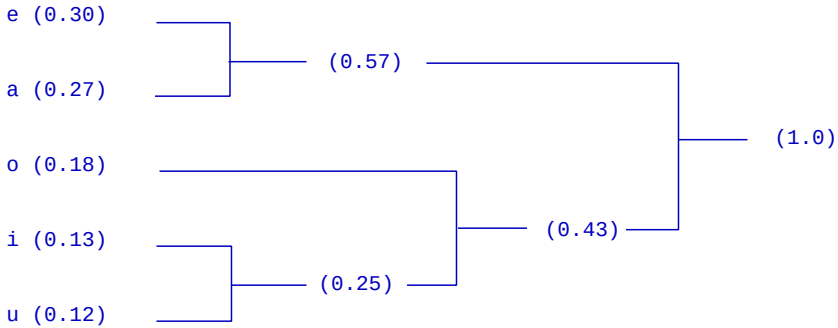
```
e (0.30)

a (0.27)

o (0.18)

i (0.13)

u (0.12)
```

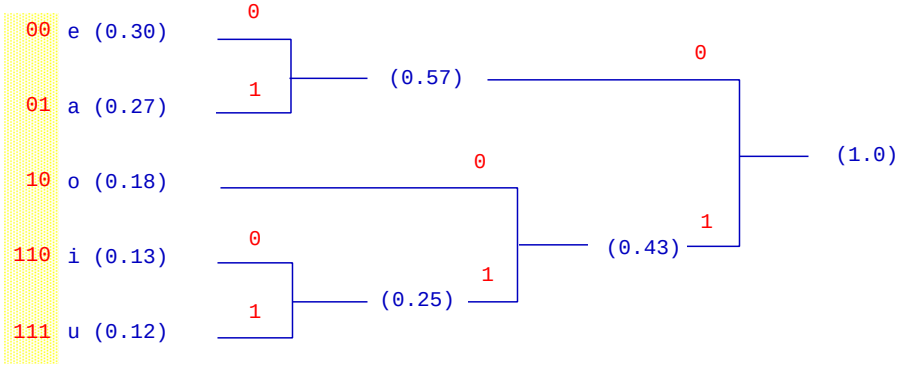
Avance automático


e (0.30)
a (0.27)
o (0.18)
i (0.13)
u (0.12)
(0.25)


e (0.30)
a (0.27)
o (0.18)
i (0.13)
u (0.12)
(0.25)
(0.43)


e (0.30)
a (0.27)
(0.57)
o (0.18)
i (0.13)
u (0.12)
(0.25)
(0.43)


e (0.30)
a (0.27)
o (0.18)
i (0.13)
u (0.12)
(0.57)
(0.25)
(0.43)
(1.0)



Otra vez

Los códigos obtenidos son:

- c(a) = 01
- c(e) = 00
- c(i) = 110
- c(o) = 10
- c(u) = 111

Ejercicio 33.

(a) ¿Cuál es, en promedio, la longitud de los códigos de Huffman generados a partir de un texto si se usan los códigos obtenidos en el ejemplo anterior?

(b) ¿Cómo es de eficiente su codificación en Morse comparada con la de Huffman?

La codificación de Huffman es instantánea, esto es ningún código es un prefijo de otro. La ventaja de las codificaciones instantáneas es que pueden decodificarse sin tener que procesar todo el código.

Por ejemplo: $c(a) = 0$ y $c(b) = 0001$ produce códigos unívocamente decodificables, pero si se encuentra un 0, es preciso seguir leyendo para saber si el 0 representa una a o el principio de una b.

Por contra, la descompresión de códigos de Huffman requiere conocer el libro de códigos utilizado, que debe bien incluirse en el fichero comprimido, bien construirse adaptativamente.

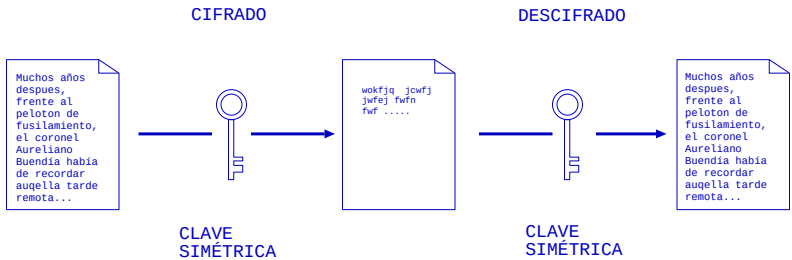
El algoritmo `bzip2` realiza una trasformación de Burrows-Wheeler (una especie de ordenamiento reversible) antes de aplicar la compresión de Huffman

5. Cifrado

El cifrado pretende que el contenido de cada mensaje sólo pueda ser desvelado por su destinatario. La desconfianza que genera la seguridad de las transacciones digitales puede se combatida con el desarrollo de sistemas de cifrado más seguros.

5.1. Cifrado con clave simétrica

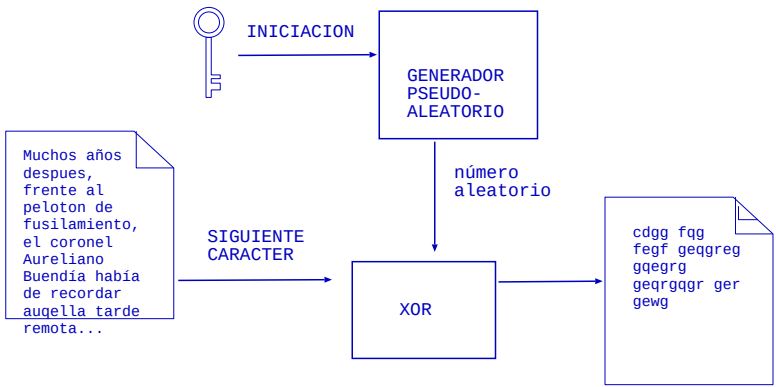
Los métodos tradicionales de cifrado se basan en enviar los mensajes a través de un canal seguro, normalmente, porque sólo el remitente y el destinatario conocen el procedimiento seguido para el cifrado.



Los métodos de cifrado en los que el remitente y el destinatario usan la misma clave se denominan métodos de clave simétrica. Los métodos de clave simétrica son sencillos pero vulnerables.

Un método de cifrado simétrico ya era utilizado en la época romana: escribiendo sobre una cinta enrollada sobre un bastón el destinatario necesitaba un bastón del mismo diámetro para leer el contenido de la cinta.

Un ejemplo de cifrado simétrico más moderno se representa a continuación:

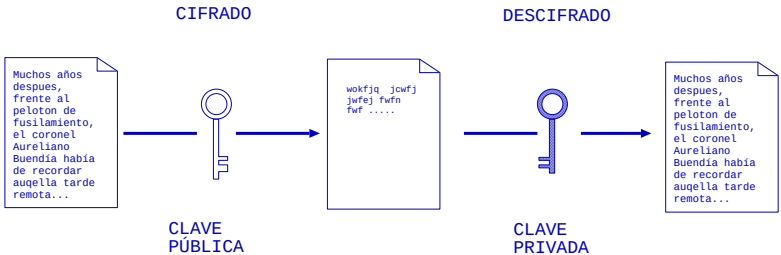


Test. ¿Cuántas claves simétricas son necesarias, para garantizar el secreto de las comunicaciones entre n personas?

(a) Una. (b) n. (c) $2n$. (d) $n(n-1)/2$.

5.2. Cifrado con clave pública

Para evitar la proliferación de claves (y el consiguiente aumento de la vulnerabilidad) se utilizan los métodos de clave pública. En ellos, el procedimiento de cifrado es público pero el de descifrado es conocido sólo por el destinatario.



¿Cómo es posible que, siendo el método de cifrado conocido, sea imposible conocer el original a partir del cifrado? Piensa en el fichero de contraseñas ("passwords") de `Linux` en `/etc/shadow`.

Algunos procesos comunes son mucho más difíciles en realizar en un sentido que en el contrario: por ejemplo, disolver un azucarillo, batir un huevo, o la jugada inicial del billar americano. Un procedimiento de "batido" sencillo de un número M es el siguiente:

$$C = M^e \mod n$$

Si el exponente e y el módulo n se eligen adecuadamente:

- Cada mensaje $M < n$ produce un cifrado C diferente.
- Es casi imposible saber qué mensaje M ha producido el cifrado C.

Esta es la base del popular algoritmo RSA (Rivest, Shamir y Adleman, 1978).

Ejercicio 34.

(a) Escribe el código cifrado $C = M^e \mod n$ de los números del 1 al 32 con $(n, e) = (33, 3)$. Comprueba que cada entrada M tiene una salida C distinta.

(b) Comprueba que la operación $C^d \mod n$ con $(n, d) = (33, 7)$ es la inversa del cifrado anterior.

(c) Si $M = 55$ y $e = 3$, el inverso es $d = 27$. ¿Podemos descifrar $C = 50$ usando funciones de la biblioteca estándar como `pow(C,d)`?

Para que RSA tenga las propiedades descritas, n debe ser el producto de dos número primos grandes (por ejemplo, de 100 cifras cada uno). Además hay que elegir los dos exponentes e y d que forman parte de la clave pública (n, e) y de la clave privada (n, d) de acuerdo con el siguiente procedimiento:

1. Elíjanse dos números primos grandes p y q.
2. Calcúlese $n = pq$ y $\Phi(n) = (p-1)(q-1)$.
3. Elíjase un impar pequeño e sin factores comunes con $\Phi(n)$.
4. Elíjase el exponente d de forma que $ed \mod \Phi(n) = 1$
5. Publíquese (n, e) y guárdese en secreto (n, d).

Para resolver la ecuación $ed \mod \Phi(n) = 1$ puede utilizarse el algoritmo de Euclides.

- **Demostración del algoritmo RSA**

Elige dos primos: 3 y elige un impar: 3

Clave pública:	
Clave privada:	

Escribe un número: $M =$ 1
Número cifrado: $C =$ 1
Escribe un número: $C =$ 1
Número descifrado: $M' =$ 1

Para el intercambio seguro de mensajes cifrados sólo es necesaria una clave pública por cada destinatario. Esta clave puede ser conocida por todo el mundo.

Los sistemas de clave pública son seguros pero demasiado lentos para mensajes largos (que deben fraccionarse). Por ello, en la práctica (p. ej., SSL) se utiliza un sistema mixto: se cifra el mensaje con una clave simétrica y se envía la clave cifrada mediante RSA.

5.3. Firma digital

Los sistemas de clave pública pueden ser utilizados no sólo para el cifrado sino también para la firma digital. Si el remitente cifra un texto (o por eficiencia, un resumen) con su clave privada, el destinatario puede leerla usando la clave pública. El truco está en que sólo el remitente puede generar un cifrado C legible con su clave pública.

Los sistemas de clave pública presentan un punto débil: ¿cómo podemos estar seguros de que la clave pública que tenemos no la ha generado un impostor?

5.4. Certificados

Los certificados digitales son análogos a los certificados tradicionales: una autoridad certificadora en quien hemos depositado nuestra confianza nos garantiza que ha comprobado que la persona o entidad certificada es quien dice ser. El certificado consta al menos de

1. una clave pública y los datos de su propietario;
2. la firma digital de la autoridad;
3. un número de serie y las fechas de validez del certificado;

La entidad certificada posee la clave privada.

La fiabilidad del certificado depende:

1. de la autoridad certificadora (del rigor de las comprobaciones que realiza para emitir un certificado, la seguridad con que guarda sus claves, etc.);

2. de la forma en que se nos haya hecho llegar el certificado.

Un certificado nos puede llegar por distintas vías: directa (por ejemplo, generado por el administrador de nuestro sistema o incrustado en el navegador que hemos adquirido) o indirecta (certificado a su vez por otra agencia certificadora). Hay una jerarquía de autoridades que se certifican unas a otras (cuya raíz ocupan empresas como VeriSign que certifica, por ejemplo, a la AEAT).

Como en el caso tradicional, la fiabilidad de un certificado es simpre relativa y debe valorarse en función del riesgo de la operación que vamos a realizar.

```
Certificate:
    Data:
        Version: 1 (0x0)
        Serial Number: 7829 (0x1e95)
        Signature Algorithm: md5WithRSAEncryption
        Issuer: C=ZA, ST=Western Cape, L=Cape Town, O=Thawte Consulting cc,
                OU=Certification Services Division,
                CN=Thawte Server CA/Email=server-certs@thawte.com
        Validity
            Not Before: Jul  9 16:04:02 1998 GMT
            Not After : Jul  9 16:04:02 1999 GMT
        Subject: C=US, ST=Maryland, L=Pasadena, O=Brent Baccala,
                 OU=FreeSoft, CN=www.freesoft.org/Email=baccala@freesoft.org
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (1024 bit)
                Modulus (1024 bit):
                    00:b4:31:98:0a:c4:bc:62:c1:88:aa:dc:b0:c8:bb:
                    33:35:19:d5:0c:64:b9:3d:41:b2:96:fc:f3:31:e1:
                    66:36:d0:8e:56:12:44:ba:75:eb:e8:1c:9c:5b:66:
                    ....
                Exponent: 65537 (0x10001)
    Signature Algorithm: md5WithRSAEncryption
        93:5f:8f:5f:c5:af:bf:0a:ab:a5:6d:fb:24:5f:b6:59:5d:9d:
        92:2e:4a:1b:8b:ac:7d:99:17:5d:cd:19:f6:ad:ef:63:2f:92:
        ....
```

Un fragmento de certificado digital.

6. Simulación computacional

La simulación computacional es una herramienta poderosa que permite predecir el comportamiento de sistemas complejos con un mínimo de análisis teórico y de inversión económica. Su uso requiere conocimientos sólidos de teoría de probabilidades y de estadística.

La simulación aleatoria se usa tanto para estudiar sistemas con comportamientos aleatorios (tráfico, economía, colas, poblaciones, etc.) como para integrar funciones y resolver ecuaciones diferenciales.

6.1. Simulación de una cola

Uno de los problemas en los que la simulación computacional es útil es el estudio de colas, por ejemplo, la cola de impresión de un sistema informático.

> El parámetro más importante en el estudio de una cola es su carga, que se define como la razón entre el tiempo promedio de servicio y el tiempo promedio entre llegadas.

Ejercicio 35. Un médico dedica un promedio de 10 minutos a cada paciente, por lo que decide citar a cada uno 10 minutos después que al anterior.

(a) ¿Cuál es el tiempo promedio de servicio?

(b) ¿Cuál es el tiempo promedio entre llegadas?

(c) ¿Cuál es el tiempo que esperará, en promedio, el paciente número t?

(d) ¿Qué tiempo debe dejarse entre las llegadas para que el tiempo de espera promedio no sobrepase los 5 minutos para ningún paciente?

Observaciones:

1. Dado que sólo se nos dice el tiempo que en promedio dedica el médico a cada paciente, debemos hacer alguna suposición para determinar el tiempo dedicado a cada uno. Una muy sencilla, es elegir un valor aleatorio entre 0 y 20 con probabilidad uniforme.

2. La mayoría de los lenguajes de programación incluyen funciones que generan un número aleatorio en el intervalo $[0, 1[$ con densidad de probabilidad uniforme. Por ejemplo, podemos usar `random()/RAND_MAX` en `C++` o `Math.random()` en `JavaScript` y `Java`.

En una cola estacionaria, si A_t es el tiempo entre la llegada t y la $t-1$, S_t el tiempo de servicio para la llegada t y W_t el tiempo que debe esperar t para ser atendido, se cumple que

$$W_t = \begin{cases} W_{t-1} + S_{t-1} - A_t & \text{si } W_{t-1} + S_{t-1} > A_t \\ 0 & \text{en caso contrario} \end{cases}$$

Además, la carga ρ es el cociente entre el promedio de S_t y el de A_t.

A continuación se representan gráficamente algunos resultados obtenidos para el tiempo de espera en la cola del ejercicio 35.

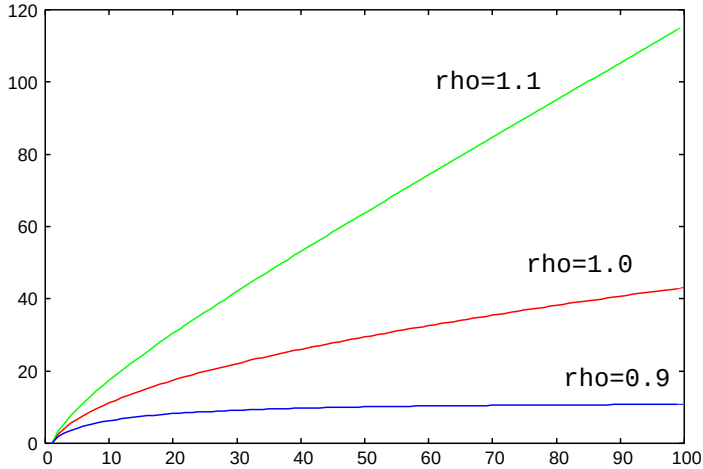


Figura 6: Tiempo de espera W_t en la cola para tres valores de ρ distintos.

Una cola con carga $\rho < 1$ es estable, es decir, el tiempo de espera tiende a un límite. En cambio, si $\rho \geq 1$ la cola no es estable y el tiempo promedio de espera crece indefinidamente.

Problema 5. ¿Es la lotería justa o injusta? Supongamos que cada jugador apuesta proporcionalmente a su renta y que todo lo apostado se lo lleva el ganador (con probabilidad proporcional a su apuesta). ¿Cómo evoluciona a largo plazo la renta de los jugadores? Implementa un programa en `Java` para estudiar el resultado y elabora un informe explicando qué hipótesis has hecho para la simulación.

6.2. Generación de números alatorios

Un número aleatorio es un número cuyo valor no se puede predecir. Por tanto, los números aleatorios no se pueden generar mediante un algoritmo.

Test. ¿Son las cifras del número π una secuencia aleatoria?

(a) Sí. (b) No.

Los auténticos números aleatorios sólo aparecen en sistemas caóticos y cuánticos. En la práctica se usan generadores pseudoaleatorios.

Un generador pseudoaleatorio sencillo es

$$I_k = (aI_{k-1} + c) \mod m$$

P. ej., `drand48()` de Linux usa aritmética de 48 bits con

$$\begin{aligned} m &= 2^{48} \\ c &= \texttt{0xB} \\ a &= \texttt{0x5DEECE66D} \end{aligned}$$

El valor I_0 que se usa como semilla (en `srand48`) debe ser un entero grande. Entonces, el cociente I_k/m proporciona una distribución uniforme en el intervalo $[0, 1[$ cuya densidad de probabilidad es $f(x) = 1$.

aunque los generadores lineales como el anterior son rápidos y eficientes:

1. un número muy pequeño va siempre seguido de otro número también pequeño;

2. los valores obtenidos no se distribuyen uniformemente en un espacio de k dimensiones.

> Los generadores más modernos (como `random` de Linux) usan una combinación no lineal de un subconjunto de números aleatorios anteriores. Ojo: !Inventarse un generador propio es peligrosísimo!

Ejercicio 36. ¿Cómo se puede simular el resultado de lanzar un dado utilizando un generador aleatorio uniforme en $[0, 1[$?

6.3. Generación de distribuciones de probabilidad

Para simular el resultado de lanzar un dado, podemos utilizar un generador uniforme en $[0, 1[$ tal como se ilustra a continuación.

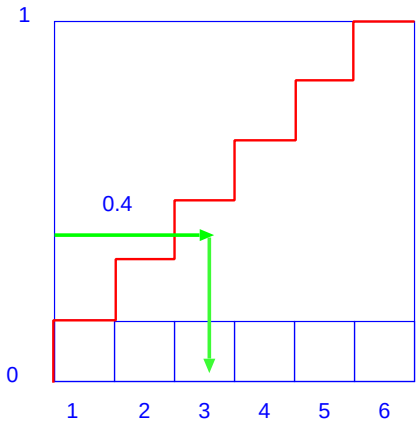


Figura 7: Representación gráfica de la generación pseudoaleatoria del experimento de lanzar un dado. En este ejemplo, el valor ζ =0.4 genera un 3 como resultado.

El método más sencillo para generar una densidad de probabilidad $f(x)$ es obtener un primitiva $F(x)$ de la función de densidad y aplicar la transformación $x = F^{-1}(\zeta)$ al valor ζ procedente de un generador uniforme.

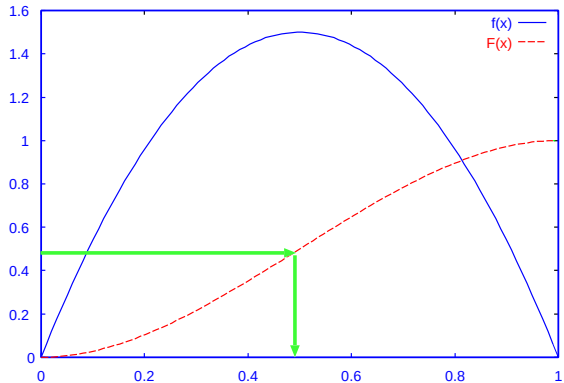


Figura 8: Función de distribución $F(x)$ asociada a $f(x) = 6x(1-x)$ y representación gráfica del cambio de variable.

Ejercicio 37.

(a) Las primitivas de $f(x) = 6x(1-x)$ tienen la forma $F(x) = 3x^2 - 2x^3 + C$. ¿Cuál es el valor de C más adecuado?

(b) ¿Cuál es la función inversa F^{-1} de la primitiva anterior $F(x)$?

En general, es costoso obtener la función primitiva inversa F^{-1}. Por ello se suele recurrir a aproximaciones numéricas o a la discretización de los resultados.

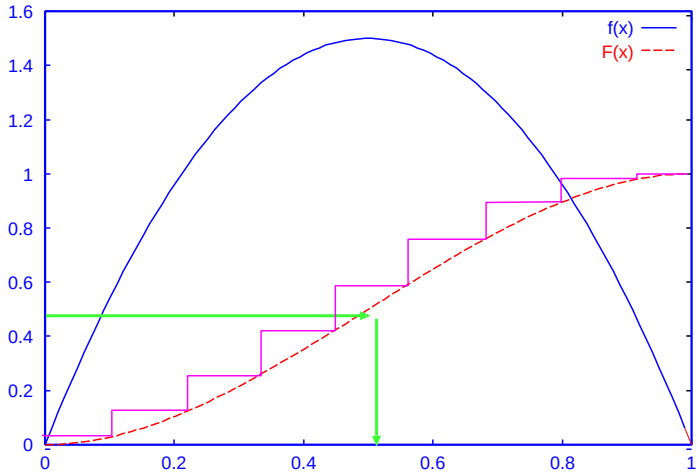


Figura 9: Función de distribución $F(x)$ discretizada a intervalos de anchura 0.1.

Por ejemplo, en una cola en la que las llegadas aleatorias son independientes unas de otras, la distribución de probabilidad que sigue A_t es la de Poisson:

$$f(t) = \frac{1}{\lambda} e^{-t/\lambda}$$

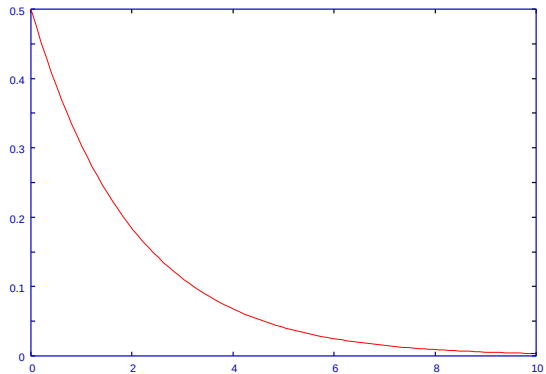


Figura 10: La distribución de Poisson con $\lambda = 2$. Nótese que la probabilidad disminuye con el tiempo, aunque no son improbables las llegadas espaciadas un tiempo muy superior a la media.

Ejercicio 38.

(a) Demuestra que λ es el tiempo promedio entre llegadas que siguen la ley de Poisson.

(b) Escribe la transformación que debe aplicarse al generador uniforme ζ para obtener una variable t con distribución de Poisson.

6.4. Margen de error en la simulación

Un estimador es una función aleatoria cuyo valor esperado coincide con el de la magnitud que se quiere medir.

En particular, sea X una variable aleatoria, x_k el resultado de la extracción k de X y φ una función. Un estimador de $\mathrm{E}[\varphi]$ es:

$$\Phi = \frac{1}{N} \sum_{k=1}^{N} \varphi(x_k)$$

Por ejemplo, en una cola podemos definir la función φ que da el tiempo de espera $W^{[t]}$ del paciente t como

$$\varphi(x) = \begin{cases} x & \text{si } x > 0 \\ 0 & \text{en caso contrario} \end{cases}$$

Esta función depende del resultado de la variable aleatoria $X^{[t-1]} = W^{[t-1]} + S^{[t-1]}$ que, a su vez, puede considerarse función de la variable $X^{[t-2]}$, etc.

Ejercicio 39.

(a) Demuestra que $\mathrm{E}[\Phi] = \mathrm{E}[\varphi]$.

(b) Demuestra que $\mathrm{Var}[\Phi] = \frac{1}{N}\mathrm{Var}[\varphi]$.

El margen de error cometido en una estimación mediante simulación aleatoria viene dado aproximadamente por:

$$\epsilon \simeq \sqrt{\mathrm{Var}[\Phi]} = \sqrt{\frac{\mathrm{Var}[\varphi]}{N}}$$

Para disminuir el margen de error en una simulación es preciso aumentar el número de experimentos N o disminuir la varianza de los resultados.

EJERCICIO 40.

(a) ¿Cuál es el margen de error cometido al calcular el número π de la siguiente forma: genérense 1000 pares aleatorios (ζ_1, ζ_2)en $[0, 1[\times[0, 1[$, calcúlese la fracción de ellos que satisface $\zeta_1^2 + \zeta_2^2 \leq 1$ y multiplíquese por 4.

(b) ¿Cuántos pares deben generarse en el método anterior para tener una precisión cercana a ϵ =0.001?

6.5. Técnicas de Monte Carlo

Monte Carlo fue una contraseña usada en Los Álamos durante el desarrollo de la bomba atómica. Ahora denomina a un gran grupo de técnicas de simulación que utilizan generadores aleatorios.

El método de rechazo para simular la densidad $f(x)$ consiste en generar aleatoriamente pares (ζ_1, ζ_2) con $\zeta_1 \in I$ y $\zeta_2 \in [0, M]$ y producir $x = \zeta_1$ si $f(x) < M\zeta_2$.

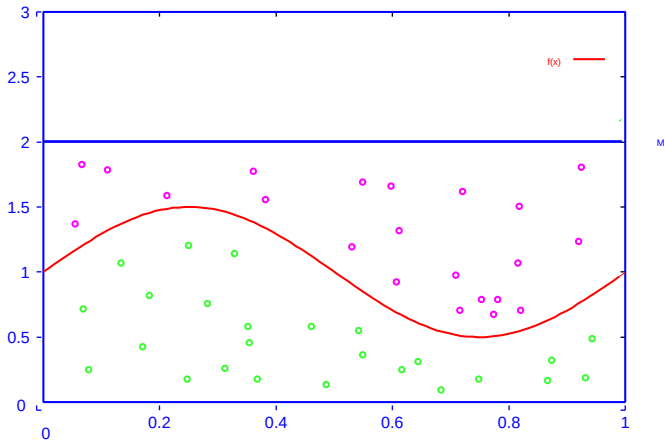


Figura 11: Generación mediante rechazo. Sólo los puntos bajo la curva $y = f(x)$ son utilizados (se selecciona su coordenada x).

El baño caliente es una combinación de los métodos de rechazo y de generación exacta mediante primitivas: una densidad g similar a f se multiplica por una constante M y se usa como "techo" de f. Así se disminuye el número de rechazos.

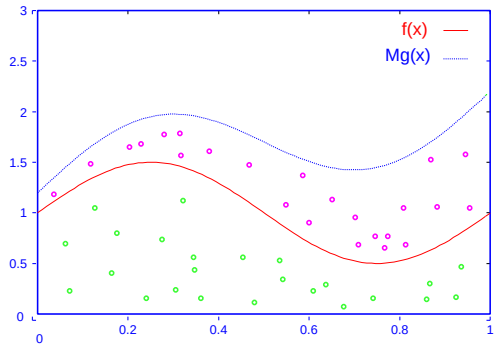


Figura 12: Generación mediante baño caliente. Sólo los puntos bajo la curva $y = f(x)$ son utilizados.

Otra variante del baño caliente consiste en asignar un peso $\xi(x) = \frac{f(x)}{g(x)}$ a cada valor x generado con densidad g. El promedio de estos pesos debe ser aproximadamente 1 y el estimador utilizado es

$$\Phi = \frac{\sum_k \xi(x_k)\varphi(x_k)}{\sum_k \xi(x_k)}$$

Ejercicio 41.

(a) Test. Si el tiempo esperado de servicio se duplica, ¿qué ocurre con el tiempo de espera promedio en una cola?

(a) No varía.

(b) Se duplica.

(c) Aumentará a más del doble.

(b) Estudia el problema de una cola con dos impresoras que recibe trabajos aleatoriamente según una distribución de Poisson y cuyo número de páginas sigue la siguiente distribución

1	2	3	4	5	6	7	8	9	10	11	12
0.1	0.2	0.25	0.15	0.1	0.05	0.05	0.025	0.025	0.020	0.020	0.010

(c) Implementa un generador aleatorio de puntos dentro de una esfera de N dimensiones.

A. Dificultades típicas en el diseño de cotas

El ejercicio siguiente pretende desvelar algunos aspectos sutiles de las funciones de cota.

Ejercicio 42.

(a) Test. Observa el árbol del ejercicio 11. Si $Y = [b, c]$ es el intervalo de cualquier nodo descendiente de $X = [a, d]$, ¿cuáles de las siguientes desigualdades se cumplen en el árbol dibujado?

(a) $f(b, c) \leq f(a, d)$.
(b) $g(b, c) \leq g(a, d)$.
(c) $h(b, c) \leq h(a, d)$.
(d) $h(b, c) \geq h(a, d)$.

(b) ¿Cuáles de las relaciones del apartado anterior se cumplirá en cualquier problema de optimización? [Ojo: difícil]

(c) ¿Por qué para todos los elementos x de X se cumple $g(X) \geq \phi(x)$ pero no es necesariamente cierto que $h(X) \leq \phi(x)$?

Podemos resumir las propiedades de las funciones f, g y h definidas en un algoritmo de maximización mediante ramificación y poda de la siguiente forma. Dado un subdominio Y del dominio de búsqueda X:

	Obligatoria	Deseable	Inaplicable
$f(Y) \leq f(X)$	✔		
$g(Y) \leq g(X)$		✔	
$h(Y) \leq h(X)$			✔

Soluciones de los Ejercicios

Ejercicio 1(a) Obviamente, $f(0) = -1$ si $m = 0$ o si $m = 1$ y $n = 1$. En cambio, si $m = 1$ y $n \neq 1$ las reglas permiten al jugador retirar 1 ficha y ganar.

En general, si se retiran k fichas quedarán $m - k$ y el contario ganará si $f(m-k, k)$ es positivo y juega correctamente. Si para alguno de los valores de k permitidos $f(m-k, k)$ es negativo tenemos una estrategia ganadora, retirar k fichas, contra la que el rival no tiene respuesta ganadora. Por tanto,

$$f(m,n) = \begin{cases} 1 & \text{si existe } k! = n : 1 \leq k \leq \text{mín}(3, m) \wedge f(m-k, k) < 0 \\ -1 & \text{en los demás casos} \end{cases}$$

La función anterior puede programarse como sigue:

```
int f (int m, int n) {
  for ( int k = 1; k < 4 && k <= m; ++k )
    if ( k != n && f(n - k, k) < 0 ) return 1;
  return -1;
}
```

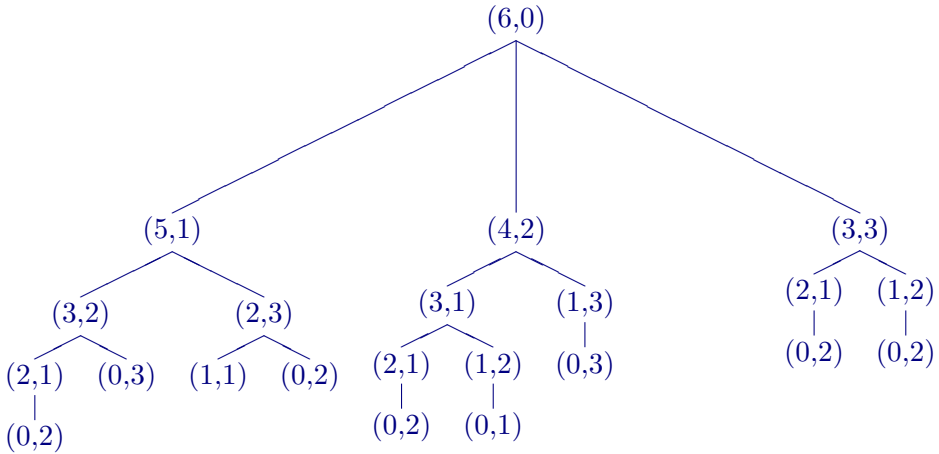
Algoritmo 3

□

Ejercicio 1(b) El algoritmo es 50 veces más lento en el segundo caso y, por tanto, inútil para valores de m que se acerquen a 100.

□

Ejercicio 1(c)



En el árbol se observa que se generan tres llamadas idénticas a la función con valor de los parámetros $n = 2$ y $m = 1$. Para calcular f para valores de n mayores, el número de repeticiones crece considerablemente.

□

Ejercicio 1(d) Una solución sencilla consiste en utilizar un almacén `A` para guardar los resultados obtenidos. De esta forma, las llamadas a la función f con parámetros idénticos se limitan a consultar el contenido $A[m][n]$ del almacén, que se calcula sólo la primera vez. La implementación puede consultarse en la página siguiente.

```
std::map<int, std::map<int, int> > A;
int f (int m, int n) {
  if ( A[m][n] == 0 ) {
    A[m][n] = -1;
    for ( int k = 1; k < 4 && k <= m; ++k )
      if ( k != n && f(m - k, k) < 0 )
        A[m][n] = 1;
  }
  return A[m][n];
}
int main(int argc, char** argv) {
  int M = atoi(argv[1]);
  for ( int m = 0; m <= M; ++m )
    for ( n = 1; n < 3; ++n )
      A[m][n] = 0;  // valor inicial
  std::cout << f (M,0) << std::endl;
}
```

Algoritmo 4

□

Ejercicio 2(a) El valor dependerá del ordenador que se esté usando para realizar el cálculo, pero típicamente será del orden de los centenares de miles. La causa de este límite es el desbordamiento de la pila de llamadas recursivas. □

Ejercicio 2(b) La función principal contiene un bucle cuyo contenido se ejecuta $3M + 3$ veces y una llamada a la función auxiliar `f`.

Dado que todo elemento del amacén `A` es distinto de cero después de la llamada a `f`(m, n) y que siempre se llama a esta función con m en el rango $[0, M]$ y n en $[1, 3]$, las instrucciones del bloque condicional no pueden ser ejecutado más de $3M + 3$ veces (una por cada vez que es cierta la condición). Además, como este bloque contiene dos llamadas recursivas, no puede haber en total más de $6M + 6$ llamadas recursivas a la función `f` (más la llamada desde la función principal).

En resumen, las instrucciones interiores del bloque condicional no pueden ejecutarse más de $3M + 3$ veces y las exteriores no pueden ejecutarse más de $6M + 7$ veces (una por llamada).

El coste espacial o memoria utilizada por el algoritmo es también proporcional a M pues tan sólo utiliza una matriz de tamaño $3M + 3$ y la pila de llamadas recursivas no necesita almacenar más de M llamadas.

□

Problema 1. Se revisará en las horas de laboratorio.

◀

Ejercicio 3. Con un algoritmo como el siguiente, la velocidad es significativamente mayor (del orden de 4 veces). Además permite calcular el resultado con un coste de tiempo y memoria viables hasta valores de m mucho mayores. La versión iterativa calcula todas las posiciones del almacén anteriores a la requerida, mientras que la versión recursiva se ahorra algunas de ellas (muy pocas) que no necesita.

```
int main(int argc, char** argv) {
  int M = atoi( argv[1] );
  std::map<long long, std::map<int, int> > A;
  A[0][1] = A[0][2] = A[0][3] = 0;
  for ( int m = 1; m <= M; ++m )
    for ( int n = 1; n < 4; ++n ) {
      A[m][n] = -1;
      for ( int k = 1; k < 4 && k <= m; ++k )
       if ( k != n && A[m - k][k] < 0 ) A[m][n] = 1;
    }
  A[M][0]=  A[M][1]>0 || A[M][2]>0 || A[M][3]>0;
  std::cout << A[M][0] << std::endl;
}
```

Algoritmo 5

Ejercicio 4. Una forma simple es modificar el programa para que haga dos pasadas por el almacén (a costa de duplicar aproximadamente el tiempo de cálculo). En la primera se determina, en orden inverso al de llenado, qué posiciones del almacén se van a necesitar. La segunda calcula las posiciones marcadas en el orden que garantiza que nunca se consultan posiciones no calculadas.

Ejercicio 4

Problema 2. Se revisará en las horas de laboratorio.

◀

Ejercicio 5(a) Tenemos dos posibilidades: borrar la u de ambas cadenas o borrar la n de ambas. En ambos casos, la subsecuencia común ("nclear" o "uclear") es de longitud 6.

□

Ejercicio 5(b) La mayor subsecuencia común es “aa”. Hay tres formas de conseguir esta subsecuencia. Una de ellas, por ejemplo, es:

```
a l g o r i t m i a
|               /
a v a n z a d a
```

□

Ejercicio 5(c) Sean dos ficheros $A = A_1 A_2 \ldots A_N$ y $B = B_1 B_2 \ldots B_M$.

La solución es trivial si alguno de los ficheros esta vacío, en cuyo caso la m.s.c. tiene longitud 0. Por tanto, si representamos mediante $f(i, j)$ el tamaño de la mayor subsecuencia común a las primeras i líneas de A y las primeras j de B, podemos escribir $f(i, 0) = f(0, j) = 0$.

Si la línea A_i es distinta de la B_j, entonces una de las dos líneas no puede ser parte de la mayor subsecuencia común y podemos escribir $f(i, j) = \text{máx}(f(i-1, j), f(i, j-1))$.

En caso de que $A_i = B_j$, la pareja (i, j) puede ser parte de la mayor subsecuencia común. Un razonamiento simple nos permite deducir que no puede haber subsecuencias comunes mayores (aunque sí iguales) que la que incluye esta pareja, así que $f(i, j) = 1 + f(i-1, j-1)$.

En resumen:

$$f(i,j) = \begin{cases} 0 & \text{si } i = 0 \vee j = 0 \\ 1 + f(i-1, j-1) & \text{si } A_i = B_j \\ \text{máx}(f(i-1, j), f(i, j-1)) & \text{en caso contrario} \end{cases}$$

□

Ejercicio 5(d) Una de las secuencias más largas es 01011000100011

□

Ejercicio 5(e) El programa recursivo calcula 252 casillas frente a 400 del recursivo (sin programación dinámica, se realizan más de 200.000 llamadas).

□

Ejercicio 5(f) En el peor caso, los costes están en $\Theta(|A||B|)$ y $\Theta(|A||B||C|)$ respectivamente.

□

Ejercicio 6(a) El problema tiene una solución trivial si $n = 0$ o si $m = 0$. En ese caso (suponiendo valores y pesos positivos), $f(n, m) = 0$. En general, el valor de $f(n, m)$ depende de qué decisión se tome sobre el objeto n. Si éste se incluye en la mochila (lo que es posible sólo si $w_n \leq m$), entonces $f(n, m) = v_n + f(n-1, m-w_n)$. En cambio, si no se toma el objeto n, entonces $f(n, m) = f(n-1, m)$. Como a priori no es posible saber cuál de estas opciones es la mejor, debemos comparar ambas y la fórmula recursiva queda:

$$f(n,m) = \begin{cases} 0 & \text{si } n = 0 \vee m = 0 \\ \text{máx}\,(f(n-1,m), v_n + f(n-1, m-w_n)) & \text{si } w_n \leq m \\ f(n-1,m) & \text{en el resto de los casos} \end{cases}$$

□

Ejercicio 6(b) La ambigüedad surge de haber expresado la complejidad en términos de dos parámetros (N y M) y no en funcion del tamaño L de la entrada. Si bien el espacio requerido para codificar los tamaños y valores de N objetos es proporcional a N, el espacio requerido para codificar el valor de M es proporcional a $\log(M)$. Por ello, la complejidad en función del tamaño de la entrada L es proporcional a $NM \simeq L\exp(L)$.

□

Ejercicio 6(c) En este caso, el valor óptimo es 39 (por ejemplo, $6 + 12 + 21$ o $6 + 33$).

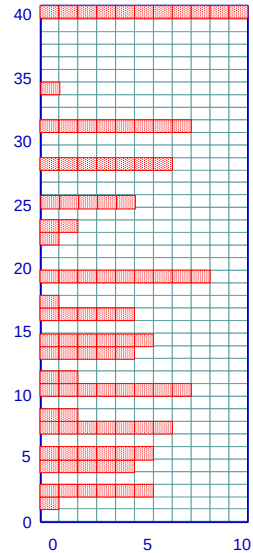


Figura 13: Parte calculada de la tabla mediante programación dinámica recursiva.

□

Ejercicio 6(d) En el caso $2^N \ll M$, el mayor coste del programa recursivo se produce en la iniciación de los valores del almacén, ya que el número de posiciones calculadas no puede ser mayor que 2^N. Para evitar este sobrecoste se debe utilizar un contenedor que permita consultar si el elemento está almacenado o no sin necesidad de crearlo (como es el caso en la implementación presentada: en el ejemplo, es perfectamente posible eliminar la inicialización). □

Ejercicio 7(a) Para su implementación en `Java` debe declararse la variable `epsilon` de tipo `static` y utilizar las funciones `Math.exp(x)` y `Math.max(x)`.

Con $\epsilon = 10^{-4}$, se realizan 131.071 llamadas.

□

Ejercicio 7(b)

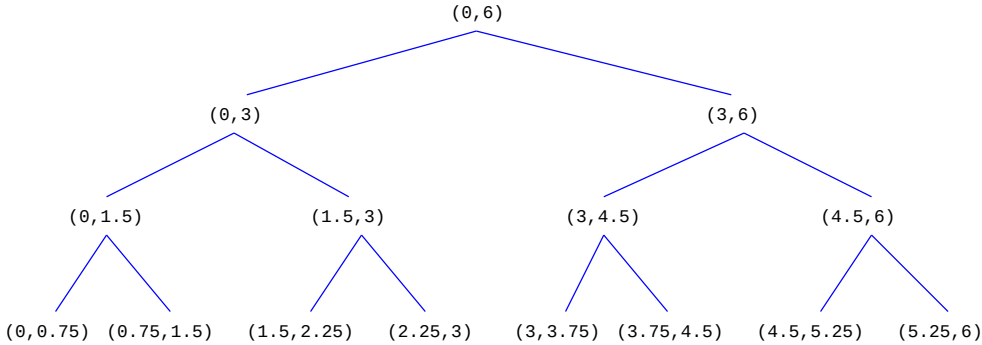


Figura 14: El árbol generado por el algoritmo.

□

Ejercicio 7(c) Es cierto, tal y como se justifica a continuación.

Sea `ybest` el mejor valor de ϕ en X, esto es, $f(X)$. Entonces existe al menos una solución óptima `xbest` $\in X$ tal que `ybest` $= \phi($`xbest`$)$. Como X es la unión de todos los conjuntos Y_k, `xbest` pertenecerá a uno de ellos, Y_n, y entonces $f(Y_n) \geq \phi($`xbest`$)$. Luego $\max_k f(Y_k) \geq$ `ybest`.

Por otro lado, `ybest` $\geq \max_k f(Y_k)$ por reducción al absurdo. En efecto, si existiese un Y_n tal que $f(Y_n) >$ `ybest`, entonces podríamos encontrar x_n tal que $\phi(x_n) >$ `ybest`. Pero como x_n está en X, entonces `ybest` no sería el valor óptimo de ϕ en X.

□

Ejercicio 8(a) En este caso, $g(3,6) = 6\exp(-3) = 0.30$.

□

Ejercicio 8(b) Dado que sea cual sea el resultado de calcular $f(3,6)$, este no puede ser superior a la cota 0.30, el máximo coincidirá con el obtenido para el intervalo $[0,3]$, esto es, 0.37.

□

Ejercicio 8(c) En este caso no es posible deducir nada, ya que la cota optimista 6 es mayor que el mejor valor 0.37, por lo que no podemos descartar la existencia de soluciones óptimas en el intervalo $[3, 6]$. Es, por tanto, más útil la cota del apartado anterior.

□

Ejercicio 9.

```
void f ( double xmin, double xmax ) {
 double xmid = ( xmin + xmax ) / 2;

 if ( xmax - xmin  < epsilon ) {
  ybest = std::max ( ybest,
               std::max ( xmin * exp(-xmin), xmax * exp(-xmax) ) );
 } else {
  if ( g ( xmin, xmid ) > ybest )
     f ( xmin, xmid );
  if ( g ( xmid, xmax ) > ybest )
     f ( xmid, xmax );
 }
}

double ybest = 0;
```

Algoritmo 6

Ejercicio 9

Ejercicio 10(a)

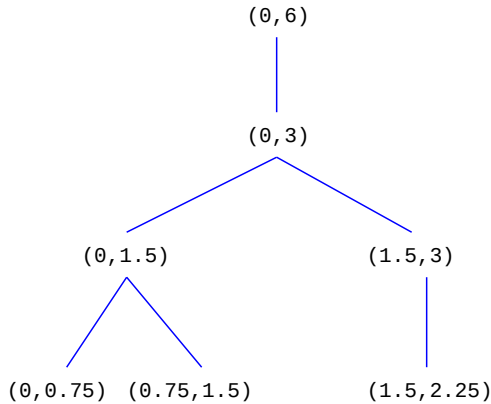


Figura 15: El árbol generado por el nuevo algoritmo.

□

Ejercicio 10(b) En el caso ϵ =1.0, no se realiza ninguna poda. Para valores de ϵ menores, la eficacia disminuye de un 50 % a un 28 %. El motivo esencial de esta pérdida de eficiencia es el retraso en encontrar un buen valor de la variable `ybest` que permita la poda eficiente.

□

Ejercicio 10(c) Para mejorar la eficiencia de la poda conviene:

1. construir mejores funciones de cota (menos optimistas);
2. elegir formas adecuadas de recorrer el árbol de estados;
3. encontrar rápidamente buenos valores para `ybest`.

El primer método (ilustrado en el ejercicio 8) depende esencialmente de nuestra maestría. Cómo cambiar la forma de recorrer el árbol tratará en la sección 2.5. El tercer método se consigue mediante el uso de funciones de cota pesimista, tal y como se verá en la sección 2.3.

□

Ejercicio 11. A continuación se representa una parte del árbol:

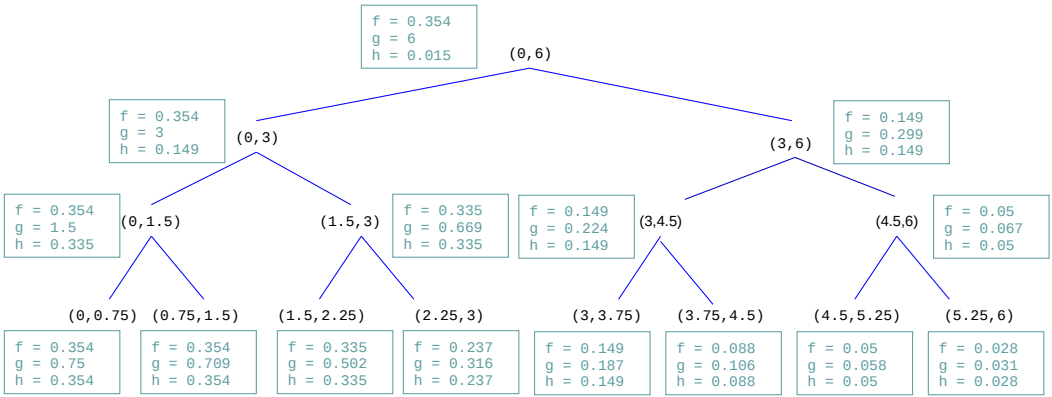


Figura 16: Valores de f, g y h y árbol de estados.

Ejercicio 12. En principio, basta con añadir ybest = std::max (ybest, h (xmin, xmax)) al principo de la función. Si se cimple que $f = h$ en los nodos hoja, podemos simplificar el algoritmo comoi sigue:

```
void f ( double xmin, double xmax ) {
 double xmid = ( xmin + xmax ) / 2;

 ybest = std::max ( ybest, h (xmin, xmax) );
 if ( xmax - xmin >= epsilon )
    if ( g ( xmin, xmid ) > ybest )
       f ( xmin, xmid );
    if ( g ( xmid, xmax ) > ybest )
       f ( xmid, xmax );
 }
}
double ybest = 0;
```

Algoritmo 7

Ejercicio 12

Ejercicio 13.

	$\epsilon = 1$	$\epsilon = 10^{-4}$	$\epsilon = 10^{-6}$
simple	15	131.071	16.777.215
con g	7	22.352	2.801.916
con h	7	1.228	14.090

Ejercicio 13

Ejercicio 14. Basta con almacenar los pares (`xbest`, `ybest`) y actualizar estos cada vez que sea necesario.

```
typedef std::pair<double, double> Result;
Result rbest ( 0, 0 );

void f ( double xmin, double xmax ) {
  double xmid = ( xmin + xmax ) / 2;

  if ( rbest.second < xmin * exp(xmin) )
      rbest = Result ( xmin, xmin * exp(-xmin) );
  else if ( rbest.second < xmax * exp (xmax) )
      rbest = Result ( xmax, xmax * exp(-xmax) );
  if ( xmax - xmin >= epsilon )
    if ( g ( xmin, xmid ) > rbest.second )
        f ( xmin, xmid );
    if ( g ( xmid, xmax ) > rbest.second )
       f ( xmid, xmax );
  }
}
```

Algoritmo 8

Ejercicio 15.

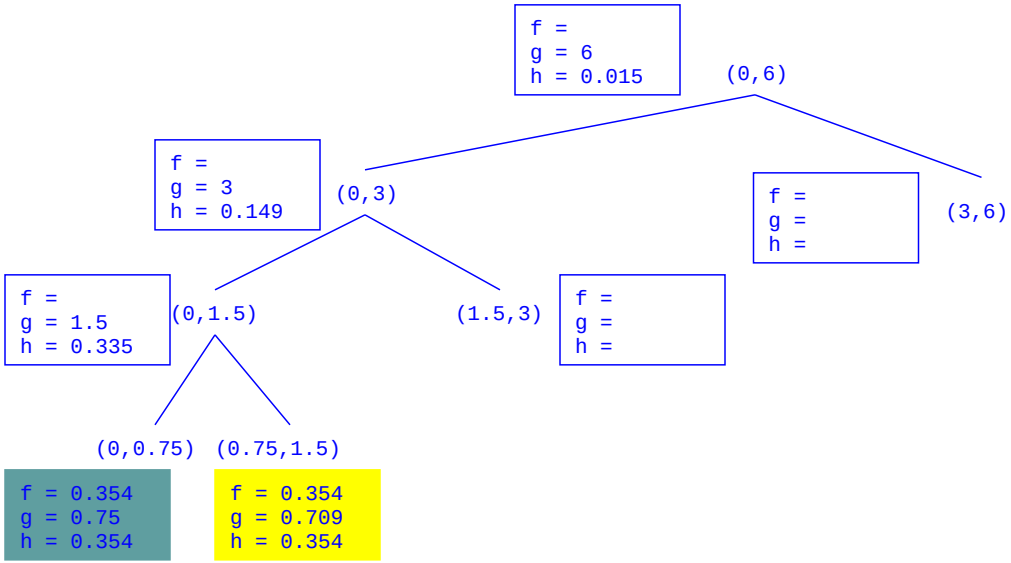


Figura 17: Nodos vivos (fondo blanco), activos (fondo amarillo) y muertos (fondo gris) en el árbol de estados cuando se está calculando $f(0.75,1.5)$.

Ejercicio 16(a) Una posible implementación en C++ es (continúa en la página sigiente):

```
typedef std::pair<double, double> Range;
typedef std::pair<double, Range> State;

double g ( Range X ) {
  return X.second*exp(-X.first);
}

double h ( Range X ) {
  return std::max( X.first*exp(-X.first),
              X.second*exp(-X.second) );
}
```

Algoritmo 9

```
void f ( Range r ) {
  std::multimap<double, Range> L;
  double  xmin, xmax, xmid, ybest = 0;

  L.insert( Result( g(r), r) );
  while ( L.size() > 0 ) {
    r = L.rbegin() -> second;    // select and
    L.erase( --L.end() );        // drop best candidate
    xmin = r.first; xmax = r.second; xmid = ( xmin + xmax ) / 2;
    ybest = std::max ( ybest, h ( xmin, xmax ) );
    if ( xmax - xmin >= epsilon ) {
      r = Range( xmin, xmid );
      if ( g(r) > rbest.first ) L.insert( State( g(r), r ) );
      r = Range(xmid, xmax);
      if ( g(r) > rbest.first ) L.insert( State( g(r), r ) );
    }
  }
}
```

Algoritmo 10

□

Ejercicio 16(b) En la implementación presentada, con $\epsilon = 10^{-6}$, el número de iteraciones es 11.421 frente a 14.090 llamadas en la mejor versión recursiva. Por tanto, la estrategia inteligente elige un recorrido aceptable y, normalmente, mejor que los ciegos diseñados por el programador. Por contra, este procedimiento consume más memoria, lo que al final, también perjudica la eficiencia temporal.

□

Ejercicio 17. Basta con añadir una instrucción del tipo

```
L.erase( L.begin(), L.lower_bound(ybest) );
```

Ejercicio 17

Ejercicio 18. Los recorridos en anchura visitan todos los nodos de un nivel antes de pasar al siguiente. Esto puede ser útil si sabemos que el árbol tiene ramas muy profundas (incluso infinitamente largas) y que la solución no está a mucha profunidad. Por contra, el número de estados vivos puede creer extraordinariamente (los árboles típicos son mucho más anchos que profundos), lo que generará problemas de memoria.

Por otra parte, no es posible utilizar recorridos en postorden, pues la poda sólo es posible si se calculan las funciones de cota antes de visitar los hijos.

Ejercicio 18

Ejercicio 19. Esto puede hacerse de varias maneras:

1. Reescribiendo los algoritmos con cuidado. Esto requiere revisar todas las operaciones de comparación y mantener dos versiones del esquema.

2. Usando el truco $\phi = -\varphi$, basado en que minimizar una función es equivalente a maximizar la de signo contrario. En este caso, es difícil interpretar siempre correctamente el significado de las funciones de cota.

3. Mantener los algoritmos y, simplemente, redefinir los operadores de comparación de soluciones ("mejor que") de forma que la comparación se haga usando la función objetivo y no directamente. Es la opción más cómoda para los programadores de `Java` o `C++`.

Ejercicio 19

Ejercicio 20. El elemento máximo de un conjunto vacío no está definido, por lo que cualquier valor que asignemos es arbitrario. Por ello, podemos elegir el que más nos interese. En los problemas habituales de maximización mediante ramificación y poda hemos visto que $f(X)$ coincide con el máximo de f entre los subconjuntos Y_k de X. Para que esto sea cierto incluso si algún $Y_n = \emptyset$ es conveniente tomar $f(\emptyset) = -\infty$, donde $-\infty$ representan un valor inferior al de cualquier solución posible.

Ejercicio 20

Ejercicio 21. Una cota pesimista viene dada por cualquier solución particular contenida en el estado. Es importante darse cuenta de que en algunos casos no existirán soluciones con las opciones $x_1, \ldots, x_m$ tomadas. En ese caso, ningún número finito ω garantiza que existe una solución mejor o igual que ω en X y la cota pesimista debe ser, $h(X) = \infty$. En caso contrario, nos conviene que $h(X)$ sea lo más baja posible y una forma que permite encontrar resultados con un coste computacional moderado es una estrategia voraz:

1. Hágase $x_i = 0$ para $i = m + 1, ...M$.

2. Sea $Q = \{m+1, ..., M\}$ y P el conjunto de trabajos tales que $x_i = 0$ para todos los trabajadores i involucrados en él, esto es, $P = \{j : \sum_i x_i T_{ij} = 0\}$.

3. Hágase Q una cola ordenada de mayor a menor productividad de cada trabajador, esto es, según el número de productos $\sum_{j \in P} T_{ij}$.

4. Elimínese el primer trabajador i de Q, hágase $x_i = 1$ y elimínenese de P todos los j tales que $T_{ij} = 1$.

5. Si $|P| > 0$ y $|Q| > 0$, vuélvase al paso 3.

6. Si $|P| > 0$ devuélvase ∞. En caso contrario devuélvase $\sum_i x_i$.

Las cotas optimistas deben garantizar que ninguna solución es mejor que el valor g dado. Una opción sencilla consiste en usar el algoritmo anterior, pero modificando los 3 últimos pasos 4 de la siguiente forma:

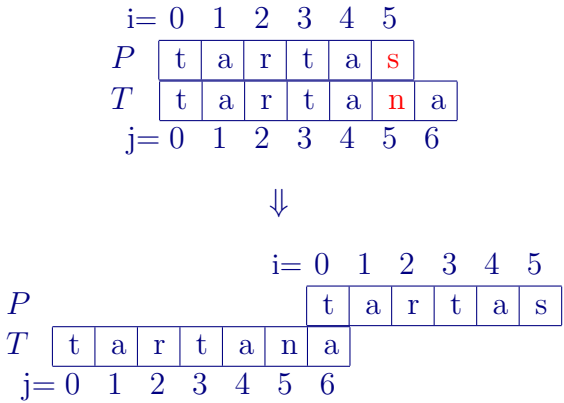
4. Hágase $n = |P|$.
5. Elimínese el primer trabajador i de Q; hágase $x_i = 1$ y $n = n - \sum_{j \in P} T_{ij}$.
6. Si $n > 0$ y $|Q| > 0$, vuélvase al paso anterior.
7. Si $n > 0$ devuélvase ∞. En caso contrario devuélvase $\sum_i x_i$.

de manera que n se calcula restando el número de trabajos en los que participa i sin tener en cuenta posibles repeticiones.

Ejercicio 21

Ejercicio 22. Para ver cuantas respuestas has acertado, aprieta el botón de Fin. Ejercicio 22

Ejercicio 23(a) Dado que la letra (en el ejemplo, la n) no se encuentra en P, podemos desplazar el principio de P hasta la posición siguiente a este carácter, esto es, incrementar j en $i + 1$.

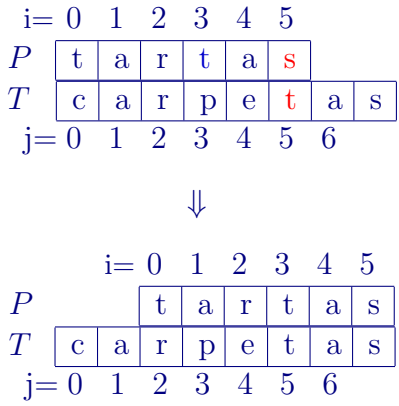


□

Ejercicio 23(b) Como al desplazar P sobre T, los nuevos caracteres aparecen por la derecha de P conviene, sobre todo si P es largo, comparar los caracteres de P y T de derecha a izquierda.

□

Ejercicio 24(a) Si la última aparición del carácter en P ocurre en la posición k (en el ejemplo $k = 3$), podemos desplazar el patrón $i - k$ posiciones:



□

Ejercicio 24(b) Evidentemente, $k = -1$ ya que entonces $i - k = i + 1$.

□

Ejercicio 25. Una implementación del desplazamiento para cada carácter es la siguiente. Primero se calcula la posición de cada carácter:

```
void
bad (string& P, int* B) {
  uint i, m = P.length();
  for ( i = 0; i < 256; ++i )
    B[i] = -1;
  for ( i = 0; i < m; ++i )
    B[ P[i] ] = i;
}
```

Algoritmo 11

Luego, el algoritmo de búsqueda de fuerza bruta se modifica como sigue:

```
void
find (string& P, string& T) {
  int i, j, m = P.length(), n = T.length(), B[256];

  bad(P, B);
  j = 0;
  while ( j + m <= n ) {
    for ( i = m - 1; i >= 0 && T[j+i] == P[i]; --i );
    if ( i < 0 ) {
      cout << "Match: " << j++ << endl;
    } else {
      j += max( 1 , i - B[ T[j+i] ] );
    }
  }
}
```

Algoritmo 12

Ejercicio 25

Ejercicio 26(a) Observa que, aunque el sufijo "ado" no vuelve a aparecer en P, el prefijo "do" de P puede hacerse casar con el final del sufijo. Por ello, la estrategia de buen sufijo tiene dos partes:

1. búsquese otra aparición más a la izquierda en P del sufijo concordante (si la nueva posición es k auméntese j en $i - k$ posiciones);

2. si no se encuentra, compruébese si un prefijo de P concuerda con un sufijo más corto (si su longitud es k, auméntese j en $m - k$);

3. si se fracasa en ambas búsquedas, increméntese j en m posiciones.

□

Ejercicio 26(b) Sólo es preciso sustituir máx$(1, B[T[j+i]])$ por

```
max ( G[i+1], i - B[ T[j+i] ] )
```

y G se calcula como en la página siguiente. El vector auxiliar $F[i]$ representa la longitud del mayor prefijo $P[i]...P[i+F[i]-1]$ que es un sufijo de P.

```
void good (string& P, int*G) {
  int m = P.length(), *F = new int (m);
  for ( i = 0; i < m; ++i ) {
     F[i] = 0; G[i] = m;
  }
  for ( i = m - 2; i >= 0; --i ) {
      for ( k = 0; k <= i && P[i-k] == P[m-1-k]; ++k );
      while ( k > 0 ) {
         F[ i - k + 1 ] = max(k, F[i-k+1] );
         --k;
      }
  }
  for ( i = 0; i < m; ++i )
     G[ P.length - F[i] ] = P.length - i - F[i];
  if ( F[0] > 0 )
       for ( i = 0; i < m; ++i )
          if( G[i] == m )
             G[i] = m - F[0];
}
```

Algoritmo 13

□

Ejercicio 27. Si suponemos que el número de palabras total de la colección será del orden de 10^8 (aunque el promedio de caracteres por palabra es cercano a 5, han de tenerse en cuanta los espacios en banco, signos de puntuación etc) y el del léxico 10^7 necesitaremos 0.4GB para guardar las direcciones y 50MB para guardar el diccionario.

Ejercicio 27

Ejercicio 28. Una forma sencilla de crear estas tablas es mediante una clase que extienda la clase TreeMap y que asocie al nombre del fichero (o a cada palabra) objetos de tipo entero. Como la clase Integer no permite operaciones sobre el contenido, puede ser conveniente definir una clase como la siguiente:

```
    class Int implements Serializable {
  int n;

  public Int (int _n)  { n = _n; }
  public int  getValue ()   { return n; }
  public void setValue (int _n) { n = _n; }
  public void inc ()       { ++n; }
  public String toString() { return "" + n; }

}
```

Ejercicio 28

Ejercicio 29. Dado el tamaño del vector, no es posible construirlo en la memoria virtual (además, las clases contenedor guardan referencias al objeto, lo que multiplica su tamaño). Por ello, hay que crear un procedimiento que acceda a bloques de 4 bytes (32 bits) del fichero.

Ejercicio 29

Ejercicio 31(a) Supongamos que la publicidad es cierta y que todos los 2^{1000} ficheros posibles de tamaño menor o igual que 1000 se comprimen en ficheros de, como mucho, 999 bits. Como $2^{999} < 2^{1000}$ sólo los 2^{999} primeros pueden dar resultados distintos y el compresor dará para el fichero número $2^{999}+1$ un resultado idéntico al de comprimir uno de los ficheros anteriores. ¿Cuál de las dos entradas devolverá el descompresor? Conclusión: la compresión debe realizarse con pérdida de información (como lo hace, por ejemplo, la compresión JPEG o MP3).

Otra forma de ver la imposibilidad de esta afirmación es que el resultado de la aplicación del algoritmo al fichero comprimido debería producir uno aún menor. Este proceso podría repetirse hasta obtener un fichero sin contenido.

□

Ejercicio 31(b) Los "bits" aleatorios pueden obtenerse escribiendo caracteres generados con valor equiprobable entre 0 y 255 en un fichero. El tamaño del fichero comprimido resultante es mayor que el original en todos los casos.

□

Ejercicio 32. Cuando el diccionario está lleno sólo hay dos posibilidades: no admitir nuevas incorporaciones o construir un diccionario nuevo (totalmente o en parte).

Ejercicio 32

Ejercicio 33(a) La longitud promedio es $0.75\times2+0.25\times3=2.25$ bits.

□

Ejercicio 33(b) Los códigos Morse de las vocales son c(a)=· – , c(e)=·, c(i)=··, c(o)= – – – y c(u)=·· – . En promedio, $0.30\times1+0.40\times2+0.30\times3=2.0$. No obstante, cada carácter codificado como Morse va separado del siguiente por una pausa (si no, no es unívocamente descodificable). Por tanto, si consideramos la pausa como un símbolo adicional el promedio sube a 3.0.

□

Ejercicio 34(a) Puedes probarlo en la página 91.

□

Ejercicio 34(b) Puedes probarlo en la página 91.

□

Ejercicio 34(c) No, debido a los errores de redondeo. Una forma sencilla de calcular de forma exacta potencias modulares $a^x \mod n$ con exponente grande es la siguiente:

```
unsigned
modpow ( unsigned a, unsigned x, unsigned n ) {
    if ( x == 1 )
       return a % n;
    else {
       unsigned r = modpow ( a, std::floor(x/2), n );
       if ( x % 2 == 0 )
         return (r*r) % n;
       else
         return (a*r*r) % n;
    }
}
```

Algoritmo 14

□

Ejercicio 35(a) Tal y como dice el enunciado, el tiempo promedio de servicio es 10 minutos.

□

Ejercicio 35(b) El tiempo entre llegadas es 10 minutos.

□

Ejercicio 35(c) Para calcular el promedio, es preciso realizar muchas simulaciones, más cuanto mayor precisión queramos en los resultados. Dado que no sabemos a priori cuantas iteraciones serán precisas para conseguir la precisión deseada, usaremos una variable `numit` cuyo valor podrá ser cambiado fácilmente. Otra variable, `tmax` contendrá el número máximo de pacientes que queremos estudiar. Por último, usaremos la variable `rho` para la carga, que en este ejemplo es 1.

Para cada paciente, el tiempo de espera es la suma del tiempo de espera más el de servicio del anterior paciente menos el tiempo entre llegadas, esto es, $W_t = W_{t-1} + S_{t-1} - A_t$.

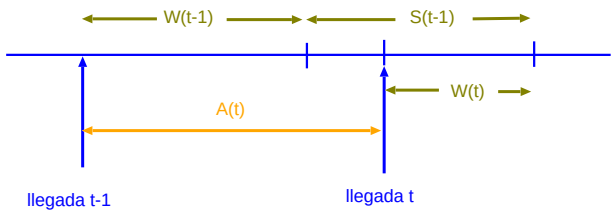


Figura 18: Representación gráfica de las relaciones temporales.

El algoritmo que simula esta cola puede encontrarse a continuación y su

implementación en JavaScript en esta dirección.

```
void simula ( double rho ) {
   int it, t;
   double *ac, A, S, W;
   ac = new double [1+tmax]; // acumulador
   ac[0] = 0;
   for ( it = 0; it < numit; ++it ) { //
     W = 0;                              // W = tiempo de espera
     for ( t = 1; t <= tmax; ++t ) {  // paciente
       A = 10;                           // A = tiempo entre llegadas
       S = rho * random()/RAND_MAX;   // S = tiempo de servicio
       W = W + S - A;                    //
       if ( W < 0 ) W = 0;               // W no puede ser negativo
       ac[t] += W;
     }
   }
   for( t = 0; t < tmax; ++t )
     cout << t+1 << ' ' << ac[t]/numit << endl;
}
```

Algoritmo 15

□

Ejercicio 35(d)

- **Demostración de una cola**

Probando con el siguiente formulario, se puede determinar que $\rho \simeq 0.83$.

Número de iteraciones	1000
Número de llegadas:	100
Tasa de ocupación:	0.5

START

□

Ejercicio 36. Sencillamente, dividiendo el intervalo en 6 partes iguales. Si el generador genera $\zeta < \frac{1}{6}$, entonces el resultado es 1; si $\frac{1}{6} < \zeta < \frac{2}{6}$ el resultado es 2; etc.

Ejercicio 36

Ejercicio 37(a) La constante C debe fijarse para que F vaya de 0 a 1 en el intervalo I donde está definida f, ya que la probabilidad total debe ser 1. Así no tendremos problemas para calcular la función inversa del número proporcionado por el generador pseudoaleatorio. En este ejemplo $I = [0, 1[$ y debemos tomar $C = 0$ para que $F(0) = 0$ y $F(1) = 1$.

□

Ejercicio 37(b) Para determinar F^{-1}, debemos resolver la ecuación de tercer grado $\zeta = 3x^2 - 2x^3$. Esta ecuación sólo tiene una solución en $[0, 1[$:

$$x = \frac{1}{2} + \cos\left(\frac{\arccos(1 - 2\zeta) - 2\pi}{3}\right)$$

□

Ejercicio 38(a) Escribiendo el valor esperado de t y usando el cambio de variable $x = t/\lambda$ obtenemos:

$$E[t] = \int_0^\infty dt\ \frac{t}{\lambda} e^{-\frac{t}{\lambda}} = \lambda \int_0^\infty dx\ x e^{-x}$$

La última integral puede calcularse fácilmente por partes y se obtiene 1.

□

Ejercicio 38(b) La primitiva de $f(t)$ es $F(t) = C - \exp(-t/\lambda)$ con $C = 1$ para que $F(0) = 0$. Por tanto, $\zeta = 1 - \exp(-t/\lambda)$ y la transformación es $t = -\lambda \log(1 - \zeta)$.

□

Ejercicio 39(a) Por la linealidad de la esperanza:

$$\mathrm{E}[\Phi] = \frac{1}{N}\sum_{k=1}^{N}\mathrm{E}[\varphi] = \mathrm{E}[\varphi]$$

□

Ejercicio 39(b) Es resultado inmediato de que la varianza es cuadrática, es decir, $\mathrm{Var}(ax + by) = a^2\mathrm{Var}(x) + b^2\mathrm{Var}(y)$ y, por tanto,

$$\mathrm{Var}[\Phi] = \frac{1}{N^2}\sum_{k=1}^{N}\mathrm{Var}[\varphi] = \frac{1}{N}\mathrm{Var}[\varphi]$$

□

Ejercicio 40(a) Es evidente que la proporción de puntos dentro del cuadrante de círculo es una cuarta parte del área del círculo de radio unidad, esto es, $\frac{\pi}{4}$. Como en este método $\varphi(x,y) = 1$ si $x^2 + y^2 < 1$ y $\varphi(x,y) = 0$ en caso contrario, φ^2 coincide con φ y

$$\int dx\ dy\ \varphi^2(x,y) = \int dx\ dy\ \varphi(x,y) = \frac{\pi}{4}$$

Por tanto,

$$\mathrm{Var}[\varphi] = \frac{\pi}{4} - \left(\frac{\pi}{4}\right)^2 \simeq 0{,}168$$

Como consecuencia de ello, para $N = 1000$ obtenemos una precisión $\epsilon \simeq 0.01$

□

Ejercicio 40(b) Para mejorar en un orden de magnitud la precisión, es preciso multiplicar por 100 el número de sucesos, esto es, $N \simeq 100000$.

□

Ejercicio 41(b) Ejercicio abierto que puede ser entregado para su revisión.

□

Ejercicio 41(c) Ejercicio abierto que puede ser entregado para su revisión.

□

Ejercicio 42(b)

1. La primera desigualdad es siempre cierta: el valor óptimo en un subintervalo Y no puede ser mayor que el óptimo en un intervalo X que lo contiene.
2. La segunda propiedad no tiene por qué cumplirse obligatoriamente (es fácil encontrar un contraejemplo), pero se cumple con frecuencia. Puede interpretarse de la siguiente manera: conviene que la cota obtenida sobre los descendientes sea mejor que la que se ha obtenido previamente sobre el nodo padre (en caso contrario, tomar la cota obtenida para el padre permitiría mejorar los resultados).
3. La tercera desigualdad no es compatible con otra propiedad deseable: si X es un nodo hoja del árbol de estados, entonces $f(X) = g(X) = h(X)$. En ese caso, se puede demostrar por inducción que $h = f$ para todos los nodos y, por tanto, la función de cota es tan difícil de calcular como la solución óptima.
4. Imponer esta última desigualdad conduce a que todas las cotas pesimistas sean peores que la de cualquier descendiente. No hay ninguna razón teórica o práctica para imponer esta restricción que conduce a valores demasiado pesimistas.

□

Ejercicio 42(c) Supongamos que el club de tenis Tenisa organiza un torneo en el que participan todos sus socios más un invitado A, que el nivel de juego de cada jugador x es $\phi(x)$ y que el mejor jugador del club es B.

Si el invitado A es mejor que el mejor jugador B del club Tenisa, esto es $\phi(A) \geq \phi(B)$, entonces A es también mejor que todos los jugadores del club. Es decir, podemos tomar como cota optimista del nivel de juego del club $g(X) = \phi(A)$, que satisface $g(X) \geq \phi(x)$.

En cambio, si A tiene un nivel de juego $\phi(A) \leq \phi(B)$, su nivel sirve como cota pesimista $h(X)$. Es decir, hay, al menos, un jugador mejor que él, pero no significa que vaya a quedar el último del torneo.

□

Soluciones de los Tests

Solución al Test:

- Como $a \leq x \leq b$, entonces $x \leq b$ y $\exp(-x) \leq \exp(-a)$. Por ser todos valores positivos, el producto de números menores es también menor, esto es, $x\exp(-x) \leq b\exp(-a)$ para cualquier $x \in [a, b]$.
- Dado que $e^{-x} \leq 1$, se satisface $xe^{-x} \leq x \leq b$.
- El resto de las funciones no garantiza una cota optimista de $f(X)$.

Final del Test

Solución al Test: Tanto $\phi(a)$ como $\phi(b)$ son soluciones en $[a, b]$ y por tanto, también la máxima de ambas.

Final del Test

Solución al Test: Aunque las cifras del número π tienen una distribución uniforme (una de las caracterísiticas de las secuencias aleatorias), pueden calcularse usando un algoritmo y no forman por tanto una auténtica secuencia aleatoria.

Final del Test

Solución al Test: Hemos visto que el tiempo de espera crece mucho más rápidamente que la carga, hasta hacerse infinito si $\rho \geq 1$.

Final del Test

Solución al Test: Las cotas optimistas mejoran al descender en el árbol. En cambio, sobre las cotas pesimistas no se puede deducir ninguna regla.

Final del Test